CPC464 Firmware

***ROM* routines and explanations**

Bruce Godden, Locomotive Software

CAS CATALOG CAS CHECK CAS IN ABANDON CAS IN CHAR CAS IN CLOSE CAS IN DIRECT CAS IN OPEN CAS INITIALISE CAS NOISY CAS OUT ABANDON CAS OUT CHAR CAS OUT CLOSE CAS READ CAS RESTORE MOTOR CAS RETURN CAS SET SPEED CAS START MOTOR CAS STOP MOTOR CAS TEST EOF CAS WRITE GRA ASK CURSOR GRA CLEAR WINDOW GRA GET ORIGIN GRA GET PAPER GRA GET PEN GRA GET W HEIGHT GRA GET W WIDTH GRA INITIALISE GRA LINE ABSOLUTE GRA LINE RELATIVE GRA MOVE ABSOLUTE GRA MOVE RELATIVE GRA PLOT ABSOLUTE GRA PLOT RELATIVE GRA RESET GRA SET ORIGIN GRA SET PAPER GRA SET PEN GRA TEST ABSOLUTE GRA TEST RELATIVE GRA WIN HEIGHT GRA WIN WIDTH GRA WR CHAR HI KL CURR SELECTION HI KL L ROM DISABLE HI KL L ROM ENBLE HI KL LDDR HI KL LDIR HI KL POLL SYNCHRONOUS HI KL PROBE ROM HI KL ROM DESELECT HI KL ROM RESTORE HI KL ROM SELECT HI KL U ROM DISABLE HI KL U ROM ENABLE IND GRA LINE IND GRA PLOT IND GRA TEST IND KM TEST BREAK IND MC WAIT PRINTER IND SCR READ IND SCR WRITE IND TXT DRAW CURSOR IND TXT OUT ACTION IND TXT UNDRAW CURSOR IND TXT UNWRITE IND TXT WRITE CHAR KL ADD FAST TICKER KL ADD FRAME FLY KL ADD TICKER KL CHOKE OFF KL DEL FAST TICKER KL DEL FRAME FLY KL DEL SYNCHRONOUS KL DEL TICKER KL DISARM EVENT KL DO SYNC KL DONE SYNC KL EVENT DISABLE KL EVENT ENABLE KL FIND COMMAND KL INIT BACK KL INIT EVENT KL LOG EXT KL NEW FAST TICKER KL NEW FRAME FLY KL NEXT SYNC KL ROM WALK KL SYNCH RESET KL TIME PLEASE KL TIME SET KM ARM BREAKS KM BREAK EVENT KM CHAR RETURN KM DISARM BREAK KM EXP BUFFER KM GET CONTROL KM GET DELAY KM GET EXPAND KM GET JOYSTICK KM GET REPEAT KM GET SHIFT KM GET STATE KM GET TRANSLATE KM INITIALISE KM READ CHAR KM READ KEY KM RESET KM SET CONTROL KM SET DELAY KM SET EXPAND KM SET REPEAT KM SET SHIFT KM SET TRANSLATE KM TEST KEY KM WAIT CHAR KM WAIT KEY LOW EXT INTERRUPT LOW FAR CALL LOW FRM JUMP LOW INTERRUPT ENTRY LOW KL FAR ICALL LOW KL FAR PCHL LOW KL LOW PCHL LOW KL SIDE PCHL LOW LOW JUMP LOW PCDE INSTRUCTION LOW PCHL INSTRUCTION LOW RAM LAM LOW RESET ENTRY LOW SIDE CALL LOW USER RESTART MC BOOT PROGRAM MC BUSY PRINTER MC CLEAR INKS MC JUMP RESTORE MC PRINT CHAR MC RESET PRINTER MC SCREEN OFFSET MC SEND PRINTER MC SET INKS MC SET MODE MC SOUND REGISTER MC START PROGRAM MC WAIT FLYBACK SCR ACCESS SCR CHAR INVERT SCR CHAR LIMITS SCR CHAR POSITION SCR CLEAR SCR DOT POSITION SCR FILL BOX SCR FLOOD BOX SCR GET BORDER SCR GET FLASHING SCR GET INK SCR GET LOCATION SCR HORIZONTAL SCR INITIALISE SCR INK DECODE SCR INK ENCODE SCR NEXT BYTE SCR NEXT LINE SCR PIXELS SCR PREV BYTE SCR PREV LINE SCR REPACK SCR RESET SCR SET BASE SCR SET BORDER SCR SET FLASHING SCR SET INK SCR SET MODE SCR SET OFFSET SCR SW ROLL SCR UNPACK SCR VERTICAL SOUND A ADDRESS SOUND AMPL ENVELOPE SOUND ARM EVENT SOUND CHECK SOUND CONTINUE SOUND HOLD SOUND QUEUE SOUND RELEASE SOUND RESET SOUND T ADDRESS SOUND TONE ENVELOPE TXT CLEAR WINDOW TXT CUR DISABLE TXT CUR ENABLE TXT CUR OFF TXT CUR ON TXT GET BACK TXT GET CONTROLS TXT GET CURSOR TXT GET M TABLE TXT GET MATRIX TXT GET PAPER TXT GET PEN TXT GET WINDOW TXT INITIALISE TXT INVERSE TXT OUTPUT TXT PLACE CURSOR TXT RD CHAR TXT REMOVE CURSOR TXT RESET TXT SET BACK TXT SET COLUMN TXT SET CURSOR TXT SET GRAPHIC TXT SET M TABLE TXT SET MATRIX TXT SET PAPER TXT SET PEN TXT SET ROW TXT STR SELECT TXT SWAP STREAMS TXT VALIDATE TXT VDU DISABLE TXT VDU ENABLE TXT WIN ENABLE TXT WR CHAR

Published by AMSOFT, a division of
Amstrad Consumer Electronics plc
Brentwood House
169 Kings Road
Brentwood
Essex

First edition 1984

Preface

The built-in ROM 'operating system' within the Amstrad CPC464 computer can be considered to be split into a BASIC interpreter and the 'firmware'. The firmware i a collection of lower level routines responsible for all the hardware driving, screen handling and real-time event handling. The current volume describes the firmware; a companion technical manual describes the BASIC language.

Our design aim at Locomotive Software was to produce the most sophisticated computer possible within the twin constraints of minimum hardware cost and a very limited timescale. We were also determined to make all the features of the machine available to a BASIC program. We recognised, however, that muct software would be written in machine code and made all the firmware routines available to an assembler programmer. It was always intended that any or all of the features within BASIC language should be simple to provide in games, applications or indeed other programming languages.

This manual was written by one member of the software team with assistance, corrections and comments (usually constructive) from the others. Thus you will find it to be accurate, authoritative and also capable of giving clear insights into the design of the machine. The manual explains not only what the firmware does, but how it does it, why it does it and what it is intended to be used for. No apology is made for the level of detail that is sometimes descended to; we are proud of our design and we hope that with this manual you will be able to use it as fully as we intended.

Locomotive Software
Dorking, Surrey
May 1984

The Contents.

1 The Firmware.

2 ROMs, RAM and the Restart Instructions.

3 The Keyboard.

4 The Text VDU.

5 The Graphics VDU.

6 The Screen.

7 The Sound Manager.

8 The Cassette Manager.

1 The Firmware.

This manual describes the firmware of the Amstrad CPC464 Microcomputer. It does not describe the BASIC language supplied with the system. The manual does describe certain aspects of the BASIC where these affect other programs and it uses BASIC as an example program when describing some features of the firmware.

The firmware is the program that resides in the lower ROM (see section 2). Its function is to control the hardware of the computer and to provide useful facilities for other programs to use. This avoids every program written having to provide its own facilities.

This manual is expected to be of interest to anyone who would like to know how the system works. It is indispensible for programmers writing machine code programs, particularly system programs (e.g. other languages) and games.

The information presented in this manual can be extremely detailed. It covers the operation of the firmware from the lowest level (e.g. driving the sound chip) to the highest level (e.g. running a queue of sounds). It is not necessary to understand all the information given to be able to use the firmware, however, a good grasp of how the system works will aid the programmer in selecting the most appropriate method for performing a particular task.

1.1 The Hardware.

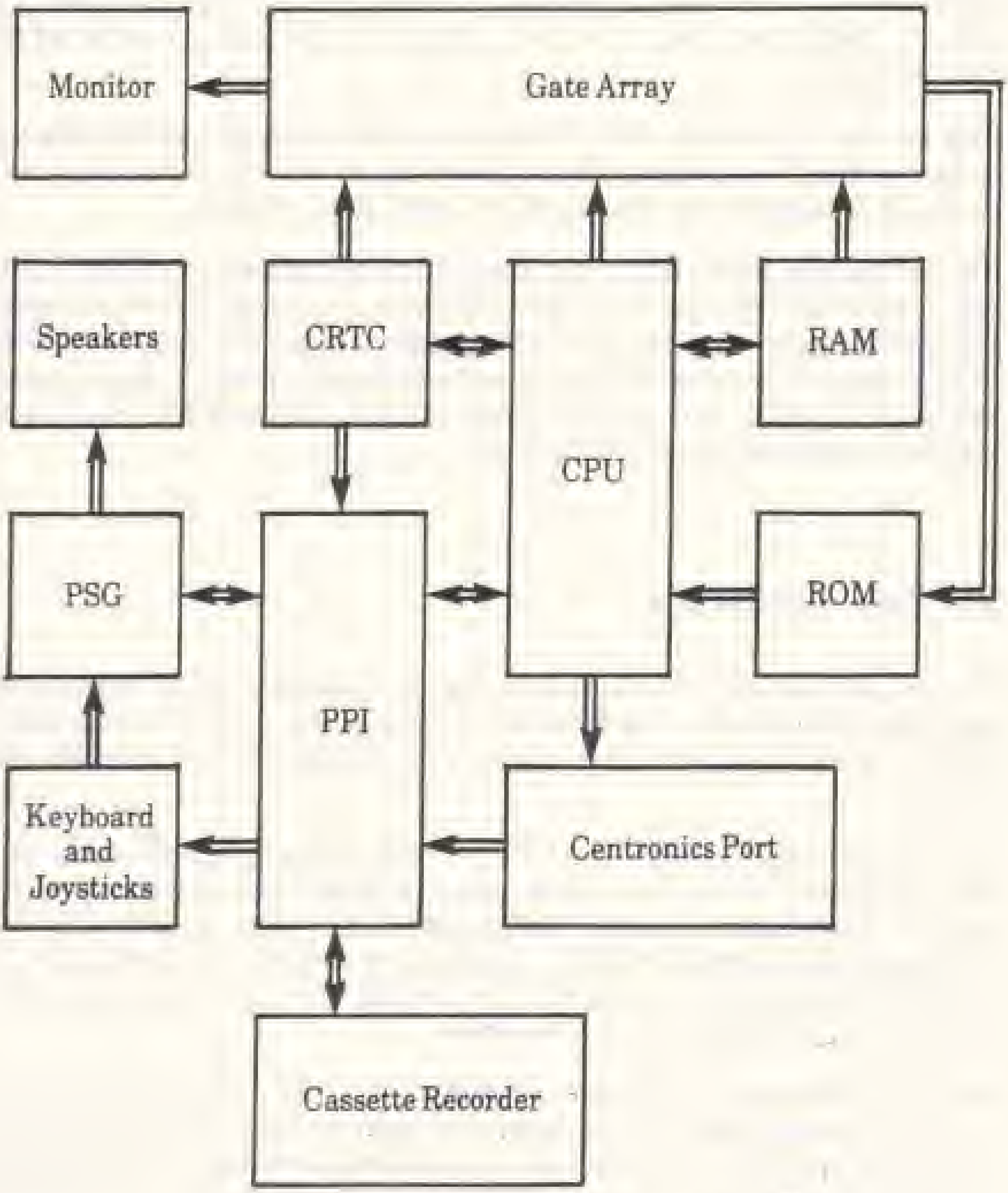
The diagram on the following page gives an indication of the different pieces of hardware in the system and how they connect to each other. For more information on how the hardware works see Appendix XII and the relevant manufacturer's data sheets.

The system centres around the CPU (Central Processing Unit) which is a Z80A microprocessor with a 4MHz clock. Next in importance is the gate array which contains miscellaneous logic to control much of the system. In particular, it controls ink colours, screen mode and ROM enabling (see section 9). In conjunction with the CRTC (Cathode Ray Tube Controller), which is a 6845 chip, the gate array generates the video signals for the monitor.

The PSG (Programmable Sound Generator) is an AY-3-8912. This chip has three channels of sound generator, a noise generator, envelope control for each channel and an I/O port. The way the sound generating hardware is used is described in section 7. The I/O port is used in input mode to sense the state of the keyboard and joystick switches.

The PPI (Parallel Peripheral Interface), which is an 8255 chip, is used to control the remainder of the system. It has three ports. Port C is used as an output port to control the cassette recorder motor, to write data to the cassette, to strobe data in or out of the PSG and to select rows of the keyboard. Port B is used as an input port to sense the frame flyback signal, the Centronics port busy signal and various option links and to read data from the cassette. Port A is used to communicate with the PSG and is set into input or output mode as required.

Accesses to memory are synchronised with the video logic - they are constrained to occur on microsecond boundaries. This has the effect of stretching each Z80 M cycle (machine cycle) to be a multiple of 4 T states (clock cycles). In practice this alters the instruction timing so that the effective clock rate is approximately 3.3 MHz.



1.2 The Division of the Firmware.

The firmware is split into 'packs' each dealing with a particular part of the system, usually a hardware device. Each pack has a section of this manual devoted to it where its operation is explained in detail. The system components and their associated packs are:

Keyboard:	Key Manager.
Screen:	Text VDU, Graphics VDU, Screen Pack.
Cassette:	Cassette Manager.
Sound:	Sound Manager.
Operating System:	Kernel, Machine Pack, Jumper.

a. Key Manager

The Key Manager is more fully described in section 3. It deals with scanning the keyboard, generating characters, function keys, testing for break and scanning the joysticks.

b. Text VDU

The Text VDU is more fully outlined in section 4. It deals with putting characters on the screen, the cursor and obeying control codes.

c. Graphics VDU

The Graphics VDU is more fully presented in section 5. It deals with plotting points, testing points and drawing lines on the screen.

d. Screen Pack

The Screen Pack is more fully detailed in section 6. It interfaces the Text and Graphics VDUs with the screen hardware and deals with aspects of the screen that affect both of these packs, such as screen mode or ink colours.

e. Sound Manager

The Sound Manager is more fully discussed in section 7. It deals with queueing, enveloping, synchronising and generating sounds.

f. Cassette Manager

The Cassette Manager is more fully explained in section 8. It deals with reading from tape, writing to tape and the cassette motor.

g. Kernel

The Kernel is more fully described in sections 2, 9, 10 and 11. It is the heart of the operating system and deals with interrupts, events, selecting ROMs and running programs.

h. Machine Pack

The Machine Pack is more fully documented in section 12. It deals with the printer and the low level driving of the hardware.

i. Jumper

Jumper, or rather, the main firmware jumpblock is listed in section 13. The entries in the jumpblock are described in detail in section 14. Jumper sets up the firmware jumpblock.

1.3 Controlling the Firmware.

The firmware is controlled by the user calling published routines rather than by the user setting the values of system variables. This will allow the firmware's variable layout to be changed in major ways without the user being affected.

The addresses of the routines the user is to call need to remain constant if the firmware is altered. This is achieved by using jumpblocks (see below).

The advantage of a routine interface is that it allows a number of different system variables to be altered by the firmware in a consistent way in one operation. If the system variables had to be set by the user then the firmware could be left in an indeterminate state if some variables had been set but not others. Also, the routine type of interface ensures that all the required side effects of a change are taken care of automatically without the user being troubled with all the details. An example of this is changing the screen mode (see section 6.1) - changing the size of the screen requires a number of other people to be informed of the change so that illegal screen positions and inks are not used.

1.4 Jumpblocks.

A jumpblock is a series of jump instructions placed in memory at well-known locations. The jumps are to the various routines in the firmware that the user might want to call. Programs that need to use the facilities provided by the routines in the jumpblock should call the appropriate jumpblock entries.

If the firmware is altered then it is quite likely that the addresses of some of the routines available to the user will change. By keeping the address of the jumpblock constant but altering the entries in the jumpblock so that they jump to the new addresses of the routines, the change is hidden from the user (providing that the user is only calling routines via the jumpblock and is not accessing the firmware directly).

To make the change to the firmware completely hidden from the user it is also necessary to keep the entry and exit conditions of the routines accessed via the jumpblock constant. The greater part of this manual is taken up with the detailed entry and exit requirements of the jumpblock entries.

The jumpblock is placed in RAM so that the user can alter the entries in it. This allows the user to trap particular entries and to substitute a new routine that will replace the standard firmware routine. Provided that the new routine obeys the entry and exit requirements of the firmware routine, the substitution will not upset programs unaware of the change.

There are four jumpblocks. These are all listed in section 13. The first and largest jumpblock is the main firmware jumpblock (see sections 13.1 and 14). This allows the user to call most firmware routines. The second jumpblock is the indirections jumpblock (see sections 13.2 and 15). The entries in this jumpblock are used by the firmware at key moments in order to allow the user to alter the action of the firmware. The last two jumpblocks are rather special. They are to do with the Kernel and allow ROMs to be enabled and routines in ROMs to be called. (See sections 13.3, 13.4, 16 and 17).

Section 1.7 below gives an example of how a jumpblock entry might be changed to alter the action of the firmware.

1.5 Conventions.

a. Notation

Processor instructions are generally referred to by their standard Z80 mnemonics. The exceptions that prove the rule are the restart instructions. The mnemonics RST 0 .. RST 7 are used rather than the more usual Z80 mnemonics RST #00 .. RST #38.

The registers are also referred to by their standard Z80 names. The flag register as a whole is referred to as F but the individual flags are called by their full name, e.g. carry. The flags are said to be true when they are set and false when they are clear. Thus a JP NC instruction would jump if carry was false and not if carry was true.

Hexadecimal numbers are indicated by prefixing the number with #, thus #7F is the number 127 in hex. All numbers not prefixed by # are in decimal.

Large numbers are often abbreviated by writing them as a multiple of 1024. For example, 32K bytes means 32 times 1024 (i.e. 32768) bytes.

b. Usage

Routines, where possible, take and return values in registers. Where more information than may be held in registers is to be passed to a routine, the address of a data area is given. The location in memory of these data areas is sometimes critical, see section 2.4.

Where a routine can succeed or fail this condition is normally passed back in the carry flag. Carry true normally implies success, whilst carry false normally implies failure.

The alternate register set, AF' BC' DE' HL', is reserved for use by the system. The user should not execute either an EX AF,AF' or an EXX instruction as these will have unfortunate consequences. (See Appendix XI for a full description.)

c. General

The logical values true and false are generally represented by #FF and #00 respectively. Often, however, any non-zero value is taken to mean true.

The bits in a byte are numbered 0..7, with bit 0 being the least significant bit and bit 7 being the most significant bit.

Where two byte (word) values are stored (in tables etc) they are always stored with the less significant byte first and the more significant byte second, unless a specific indication to the contrary is given. This is in accordance with the standard way the Z80 stores words.

Tables and the like are always laid out with byte 0 being the first byte of the table. When the address of such a table is given this is the address of byte 0 of the table unless otherwise indicated.

When the computer is turned on (or when it is reset) it completely initialises itself before running any program. This initialisation is known as early morning startup, abbreviated to EMS from now on.

1.6 Routine Documentation.

Each routine described in this manual has entry and exit conditions associated with it. Where there are other points of interest about the routine these are normally given in a section after the entry and exit conditions. Such points include whether interrupts are enabled and a fuller description of the parameters and side effects of the routine.

There are two reasons for providing this information. Firstly it tells the user what will happen when the routine is called. Secondly it tells the user what a replacement routine is expected to do.

> The entry conditions tell the caller of the routine what the routine expects to be passed to it. When calling a routine all values specified must be supplied. Values may only be left out where the routine documents that they are optional. When providing a replacement routine to fit this interface only information that is specified may be used, although not all of it need be used.
>
> The exit conditions tell the caller what values the routine passes back and which processor registers are preserved. Registers that are documented as being corrupted may be changed by the routine or may not. The user should not rely on their contents. When providing a routine to fit this interface it is extremely important that registers documented as being preserved are indeed preserved and that the values returned are compatible with the original routine. Corrupting a register or omitting a result will usually cause the system to fail, often in subtle and unexpected ways.

Often a routine will have different exit conditions depending on some condition or other (usually whether it worked or not). In these cases the specific differences in the exit conditions are given for each case and all conditions that remain the same irrespective of the case are given in a separate section (marked 'always').

There are abundant examples of routine interfaces in sections 14 to 17.

1.7 Example of Patching a Jumpblock.

The following is an example of how the jumpblocks may be used. At this stage many of the concepts introduced may be unfamiliar to the reader. However, since altering jumpblocks is an important technique for tailoring the system to a particular purpose the example is given here. Later sections will explain the actions taken here.

Suppose an assembler program is being written that is intended to use the printer when it is finished. While this program is being written it would save time and paper if the program could be made to use the screen instead of the printer. However, changing the program itself to use the screen could introduce bugs when it is changed back to using the printer. What is needed is a way of altering the action of the firmware that drives the printer - and this is what a RAM jumpblock is for.

The technique that will be used is to 'connect' the printer to a particular text window. This can be achieved by writing a short routine to send the character to the screen and patching the entry in the jumpblock for sending characters to the printer, MC PRINT CHAR, so that it jumps to this routine instead of its normal routine.

The substitute routine will have to obey the entry/exit conditions for MC PRINT CHAR. These can be found in the full description of this entry in section 14. Briefly they are as follows:

MC PRINT CHAR:

Entry conditions:

A contains character to print.

Exit conditions:

If the character was sent OK:
Carry true.
If the printer timed out:
Carry false.
Always:
A and other flags corrupt.
All other registers preserved.

The action of the substitute routine will be to select the screen stream that the printer output is to appear on, to print the character on the stream and then to restore the stream that was originally selected. To do this the substitute routine will need to call the routines TXT STR SELECT and TXT OUTPUT. Once again the full descriptions of these jumpblock entries can be found in section 14. The entry/exit conditions are as follows:

TXT STR SELECT:

Entry conditions:

A contains stream number to select.

Exit conditions:

A contains previously selected stream number.
HL and flags corrupt.
All other registers preserved.

TXT OUTPUT:

Entry conditions:

A contains character to print.

Exit conditions:

All registers and flags preserved.

The code for the substitute routine could be written as follows (stream 7 has been chosen as the stream on which printer output is to appear):

```
PUSH HL
PUSH BC
;
LD B, A                     ;Save the character to print
;
LD A, 7                     ;Printer stream number
CALL TXT_STR_SELECT         ;Select the printer stream
LD C, A                     ;Save the original stream number
;
LD A, B                     ;Get the character again
CALL TXT_OUTPUT             ;Send it to the screen
;
LD A, C                     ;Get the original stream number
CALL TXT_STR_SELECT         ;Reselect the original stream
;
POP BC
POP HL
SCF                         ;The character was sent OK
RET
```

Note the following points:

1/ MC PRINT CHAR preserves HL and BC. The routine above uses B and C for temporary storage and HL is corrupted by TXT STR SELECT. HL and BC are therefore pushed and popped to preserve them through the substitute routine.

2/ MC PRINT CHAR returns a success/fail indication in the carry flag. Since the routine above can never fail it always sets the carry flag to indicate success.

3/ The routine above does not change which text stream is selected. It selects the stream it is going to print on and restores the previously selected stream when it has printed the character. The firmware is written in such a way as to allow routines to restore the original state when they finish if required.

To use the substitute routine it is necessary to patch it into memory and to change the jumpblock entry for MC PRINT CHAR to jump to it. Assume that some memory at #AB00 has been reserved for the substitute routine and that the routine has been patched into memory. The MC PRINT CHAR entry in the jumpblock is at location #BD2B (as can be seen by inspecting section 13.1.8). The three bytes of the entry should be set to the instruction JP #AB00 by patching as follows:

```
#BD2B     #C3
#BD2C     #00
#BD2D     #AB
```

From now on all text sent to the printer will appear on the screen on stream 7. Of course, stream 7 should have its window set so that it does not interfere with any other streams using the screen.

This redirection will remain in force until the jumpblock entry is restored. This can be achieved by patching the jumpblock back again or by calling JUMP RESTORE or by causing an EMS initialisation to take place by resetting the system.

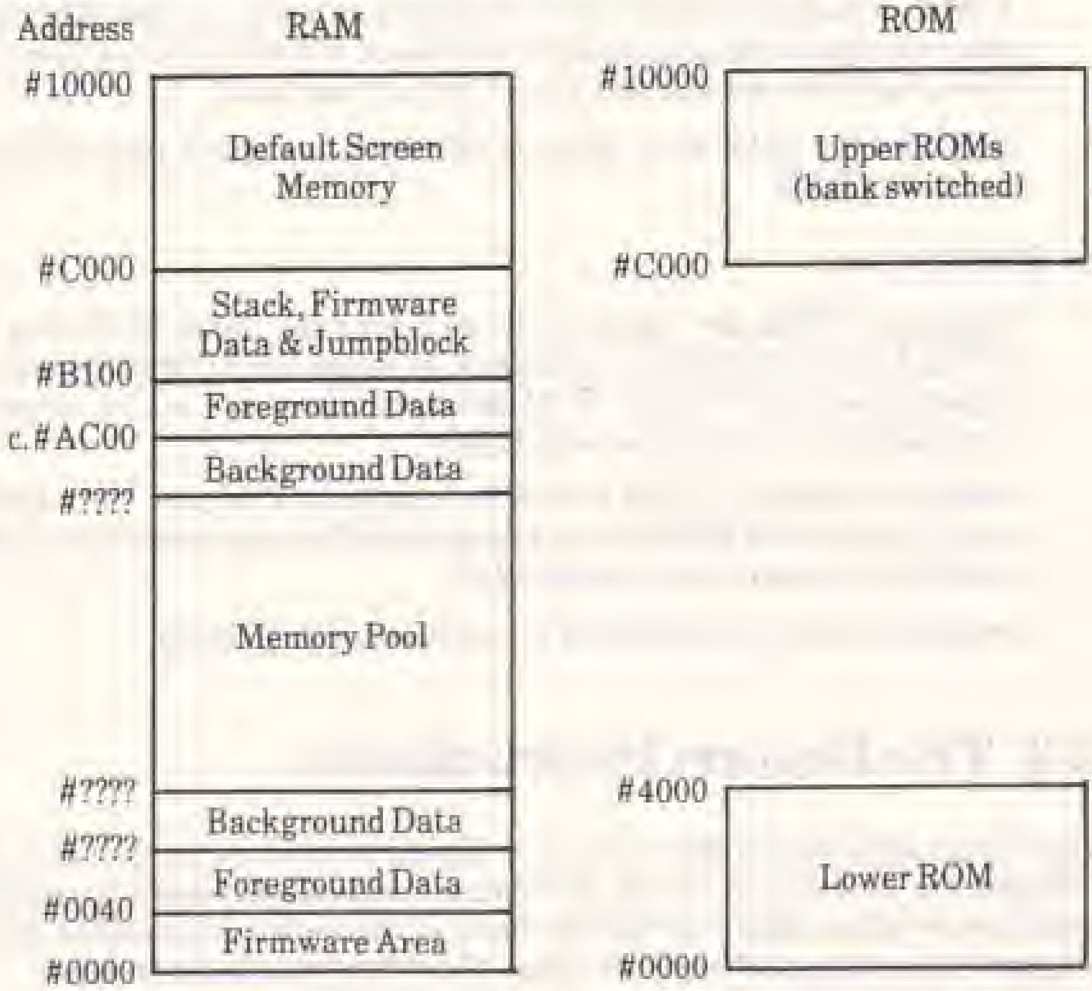
2 ROMs, RAM and the Restart Instructions.

The system has 32K of ROM and 64K of RAM in the Z80's 64K address space. To allow this the ROM can be enabled or disabled as required. Additional expansion ROMs can be selected giving up to 4632K of program area.

All the Z80 restart instructions, except for one, have been reserved for system use. RST 1 to RST 5 are used to extend the instruction set by implementing special call and jump instructions that enable and disable ROMs. RST 6 is available to the user.

2.1 Memory Map.

The memory map is complicated by the fact that into the Z80's address space of 64K bytes has been squeezed 64K bytes of RAM, 32K bytes of ROM and provision for ROM expansion of up to 252*16K (nearly 4M) bytes. The address space is divided as follows:



The sizes of the two background areas depend on the background ROMs fitted to the machine (see section 9).

The upper foreground data area need not have its lower bound at #AC00 but this is the default setting (as used by BASIC). The lower foreground data area need only be reserved if it is needed (this area is not used by BASIC and is set to zero length). The memory pool left between the background data areas is also for the foreground program to use (see section 9).

The 32K of on-board ROM is split into two sections which are handled separately. Henceforth these will be discussed as if they were separate ROMs. The firmware resides in the lower ROM. The BASIC resides in the upper ROM. This upper ROM is bank switched so that up to 252 expansion ROMs (see section 9) can replace it in the memory map.

2.2 ROM Selection.

There are two mechanisms for switching ROMs in and out of the address space:

a. ROM State.

The upper and lower ROMs may be enabled and disabled separately. When the upper ROM is enabled data read from addresses between #C000 and #FFFF is fetched from the ROM. Similarly, when the lower ROM is enabled data read from addresses between #0000 and #3FFF is fetched from the ROM. When the ROMs are disabled data is fetched from RAM.

Note that the ROM state does not affect writing which always changes the contents of RAM.

b. ROM Select.

Expansion ROMs are supported by switching the upper ROM area between ROMs. Expansion ROMs are addressed by a separate ROM select address byte implemented in I/O space. ROM select addresses are in the range 0..251, providing for up to 252 expansion ROMs.

When the machine is first turned on it selects ROM zero. This will usually select the on board ROM, but an expansion ROM may be fitted at this address, which will pre-empt the on-board ROM.

See section 9 for a description of the use of expansion ROMs.

2.3 The Restart Instructions.

The Kernel supports the store map in a number of ways. In particular a variety of facilities are provided to handle subroutine addresses extended to include ROM select and/or ROM state information. Some of the restart instructions are used to augment the existing Z80 instruction set. The other restarts are reserved.

The firmware area between #0000 and #003F is set up so that the restarts operate whatever the current ROM state is. The user should not alter the contents of this area except as indicated in section 17.

The restarts are as follows. A fuller description of their operation can be found in section 17.

a. The Extended Instruction Set.

LOW JUMP (RST 1)

RST 1 jumps to a routine in the lower 16K of memory. The two bytes following the restart are assumed to be a 'low address' - so RST 1 can be considered to be a three byte instruction, rather like a JP instruction.

The top 2 bits of the 'low address' define the ROM enable/disable state required; the bottom 14 bits give the actual address (in the range #0000 to #3FFF) to jump to once the ROM state is set up. When the routine returns the ROM state is restored to its original setting.

The firmware jumpblock, through which firmware routines should be called, makes extensive use of LOW JUMPs. These LOW JUMPs request the lower ROM to be enabled, so that the lower ROM may be disabled except when the firmware is active.

SIDE CALL (RST 2)

RST 2 calls a routine in an associated ROM. It has a very specialised use. A foreground program (see section 9) may require more than 16K of ROM. The side call mechanism allows for calls between two, three or four associated ROMs without reference to their actual ROM select addresses, provided that the ROMs are installed next to each other and in order.

The two bytes following the restart instruction give the 'side address' of the routine to call - so the RST 2 can be considered to be a three byte instruction, rather like a CALL instruction. The top 2 bits of the 'side address' specify which of the four ROMs to select; the bottom 14 bits, when added to #C000, give the actual routine address. The upper ROM is enabled, the lower ROM is disabled. Both the ROM state and the ROM select are restored to their original settings when the routine returns.

FAR CALL (RST 3)

RST 3 calls a routine anywhere in memory, in RAM or in any ROM. The two bytes following the restart are assumed to be the address of a 'far address'. The 'far address' is a three byte object, which takes the form:

Bytes 0..1:	Actual address of routine to call.
Byte 2:	ROM select/state required.

The ROM select/state byte may take the following values:

0..251:	Select the upper ROM at this ROM select address. Enable the upper ROM, disable the lower ROM.

252..255: No change of ROM select, enable/disable ROMs as follows:

252: Enable upper ROM, enable lower ROM.
253: Enable upper ROM, disable lower ROM.
254: Disable upper ROM, enable lower ROM.
255: Disable upper ROM, disable lower ROM.

Note that the 'far address' is not itself contained in the 'instruction', but is pointed at. This is because the ROM select address will depend on the particular order in which the user has chosen to install expansion ROMs and must be established at run time.

Both the ROM state and the ROM select are restored to their original settings when the routine returns.

RAM LAM (RST 4)

RST 4 reads the byte from RAM at the address given by HL. It disables both ROMs before reading and restores the state afterwards. This 'instruction' avoids the user having to put a read routine into the central 32K of RAM to access RAM hidden under a ROM.

Writing to a memory location always changes the contents of RAM whatever the ROM enable state.

FIRM JUMP (RST 5)

RST 5 turns on the lower ROM and jumps to a routine. The two bytes following the restart are assumed to be the address to jump to - so RST 5 can be considered to be a three byte instruction, rather like a JP instruction. The lower ROM is enabled before jumping to the routine and is disabled when the routine returns. The state of the upper ROM is left unchanged throughout.

b. The Other Restarts.

RESET (RST 0)

RST 0 resets the system as if the machine has just been turned on.

USER RESTART (RST 6)

RST 6 is available for the user. It could be used to extend the instruction set in the same way that other restarts have been used, or it could be used for another purpose such as a breakpoint instruction in a debugger.

Locations #0030 to #0037 inclusive in RAM may be patched in order to gain control of the restart. If the lower ROM is enabled when the restart is executed then the code in ROM at this address causes the current ROM state to be saved in location #002B. Then the lower ROM is disabled and the firmware jumps to location #0030 in RAM. If the lower ROM is disabled then the restart calls #0030 as normal for this Z80 restart instruction.

INTERRUPT (RST 7)

RST 7 is reserved for interrupts (see section 10), it must not be executed by a program.

2.4 RAM and the Firmware.

The ROM state should be transparent to the user. If the current foreground program (see section 9) is in ROM then the normal ROM state is to have the upper ROM enabled and the lower ROM disabled. If the current foreground program is in RAM then the normal state is to have both ROMs disabled. These states allow the foreground program free access to the memory pool. When a firmware routine is called the lower ROM is enabled and the upper ROM is usually disabled. This allows the firmware free access to the default screen memory (but not to all the memory pool). When the firmware routine returns the ROM state is automatically restored to what it was.

The cases where the ROM state is important are:

a. Stack

The hardware stack should never be below #4000, otherwise serious confusion will occur when the lower ROM is enabled and the stack is used - for example, when interrupts occur or the firmware is called.

Similarly, it is inadvisable to set the stack above #C000 unless it is certain that the upper ROM is never enabled when the stack is in use.

The system provides a stack area immediately below #C000 which is over 256 bytes long. This should be adequate for most purposes.

b. Communication with the firmware.

Most firmware routines take their parameters in registers. However, some use data areas in memory to pass information. Most firmware routines that use data areas in memory read these directly without using RAM LAMs (see above) or the equivalent. These routines are affected by the ROM state and the ROM select. They will read data from a ROM if the ROM is enabled and the routine is given a suitable address. (Note that the jumpblock disables the upper ROM when the firmware is called). Other firmware routines that use data areas in memory always read from RAM. They are unaffected by the ROM state and the ROM select.

Routines that always access RAM will mention this in the description of the routine. Other routines may be assumed to be affected by the ROM state. In particular the various data blocks used by the Kernel must lie in the central 32K of RAM for the Kernel to be able to use them.

c. **Communication between upper ROMs.**

Programs in upper ROMs may call routines in other ROMs, using the various Kernel facilities. There is no provision in the firmware, however, for a program in one ROM to access constants in another.

The majority of firmware routines are called via the firmware jumpblock, which starts at location #BB00, in the firmware RAM area. The Kernel routines associated with the memory map are called via one of two other jumpblock areas: the LOW area between #0000 and #003F, and the HIGH area starting at #B900. All of these routines and jumpblocks are copied out of the lower ROM into the firmware RAM area when the Kernel is initialised. Thus they work independently of the ROM state.

3 The Keyboard.

The Key Manager is the pack associated with the keyboard. All the attributes of the keyboard are generated and controlled by the Key Manager. These attributes include repeat speed, shift and control keys, function keys and key translation. The joysticks are also scanned by the Key Manager.

The Key Manager has three levels of operation. The lowest level scans the keyboard, the middle level converts the key pressings into key values and the top level converts the key values into characters. The user may access the Key Manager at whichever level is most appropriate for a given program. It is usually unwise, however, for a program to mix accesses at different levels.

3.1 Keyboard Scanning.

The keyboard is completely software scanned. This scan occurs automatically every fiftieth of a second. The keyboard hardware is read and a bit map noting which keys are pressed is constructed. This bit map is available for testing if specific keys are pressed (see KM TEST KEY). As the bit map is constructed keys that are newly pressed are noted and markers are stored in a buffer until needed. If no newly pressed keys are found then the last key pressed may be allowed to repeat if it is still down (see section 3.5). The keyboard is 'debounced' in that a key must be released for two consecutive scans before it is marked as released in the bit map. This 'debouncing' hides multiple operations of the key switch as it opens or closes.

At this stage only four keys are treated specially. The two shift keys and the control key are not stored in the key buffer themselves. Instead, when any other marker is stored the states of the shift and control keys are noted and put into the buffer as well. The escape key generates a marker as normal but may also have other effects depending on whether the break mechanism is armed (see section 3.6).

There is a problem with scanning the keyboard. If three keys at the corners of a rectangle in the key matrix are all pressed at the same time then the key at the fourth corner appears to be pressed as well. There is no way to avoid this problem as it is a feature of the keyboard hardware. All key combinations used by the firmware (and the BASIC) have been especially designed to avoid this effect.

3.2 Key Translation.

When the user asks for a key (KM WAIT KEY or KM READ KEY) the next key pressed marker is read from the key buffer. The marker is converted to a key number and this is looked up in one of three translation tables.

Which table is used depends on whether the shift and control keys were pressed when the key was pressed. One table is used if the control key was pressed, another is used if either shift key was pressed but control was not, the third is used if neither shift nor control keys were pressed. The contents of these tables can be altered by the user as required (by calling KM SET CONTROL, KM SET SHIFT and KM SET TRANSLATE respectively).

The value extracted from the table may be a system token, an expansion token or a character. Expansion tokens and characters are used by the top level of the Key Manager (see 3.3 below) and are passed up from the middle level when they are found in a table. There are three system tokens, which are obeyed immediately they are found in a table. After obeying the token the next marker is read from the buffer and translated.

The default translation tables are described in Appendix II.

The immediately obeyed System tokens are:

a. Ignore (#FF)

The key pressed is to be ignored.

b. Shift lock (#FE)

The shift lock is to be toggled (turned on if it is currently off and turned off if it is on).

c. Caps lock (#FD)

The caps lock is to be toggled (turned on if it is off and off if it is on).

3.3 Characters from the Keyboard.

When the user asks the top level for a character (KM WAIT CHAR or KM READ CHAR) a key is fetched from the middle level. If this is a character (#00..#7F or #A0..#FC) then it is passed straight back. If it is one of the 32 expansion tokens (#80..#9F) then the string associated with the token is looked up. The characters in this string are passed out one at a time with each request for a character until the end of the string is reached.

There is only one character with a special meaning at this level. This is character #EF which is produced when pressing the escape key generates a break event (see section 3.6). It has no effects, it is merely a marker for the place in the buffer where a break event was generated. It is intended to be used to allow all characters before the break to be discarded. This character is not generated by the translation tables and thus cannot be changed by altering them.

A single 'put back' character is supported. When the user puts back a character this character will be returned by the next call to the top level of the Key Manager. This is intended for use by programs that need to test the next character to be read from the keyboard without losing it (when processing breaks perhaps).

3.4 Shift and Caps Lock.

a. Shift lock

When shift lock is engaged then the keys pressed are translated as if a shift key is pressed.

The shift lock is toggled by a system token (see 3.2 above) which is normally generated by pressing CTRL and CAPS LOCK.

b. Caps lock

When caps lock is engaged then alphabetic characters read from the keyboard are converted to their upper case equivalents. This case conversion is applied before expansion tokens are expanded and so expansions are not capitalised.

The caps lock is toggled by a system token (see 3.2 above) which is normally generated by pressing CAPS LOCK (without control).

3.5 Repeating keys.

There is a table, which the user can alter as desired, that specifies which keys are allowed to repeat when held down (see KM SET REPEAT). The default setting for this table is described in Appendix III. Briefly, the default is to allow all keys to repeat except the ESC, TAB, CAPS LOCK, SHIFT, ENTER and CTRL keys and the 12 keys in the numeric keypad (the function keys).

The speed at which keys repeat and the delay before the first repeat can be set by the user (see KM SET DELAY). The default speed produces up to 25 characters a second with a 0.6 second start up delay.

A key is allowed to repeat if the following conditions are satisfied:

1/ The appropriate time has passed since the key was first pressed or it last repeated.

2/ The key is still pressed.

3/ No other key has been pressed since the key was first pressed.

4/ The key is marked as allowed to repeat in the repeat table.

5/ There are no keys stored in the key buffer.

Condition 5 above means that the repeat speed and start up delay set the maximum speed at which a key is allowed to repeat. If a program is slow about removing keys from the buffer then the generation of keys will adjust itself to this. Thus it is impossible to get a large number of keys stored in the buffer simply by holding a key pressed.

When reading or writing from the cassette the ESC key is handled in a different manner which is described in section 8.12.

3.6 Breaks.

Breaks can occur when the keyboard scanner detects that the ESC key is pressed. When the escape key is found to be pressed the indirection KM TEST BREAK is called to deal with the break. The default setting for this routine tests whether the SHIFT, CTRL and ESC keys and no others are pressed. If so then the system is reset (by executing an RST 0), otherwise the break mechanism is invoked.

If the break mechanism is disarmed then no action is taken other than the normal insertion of the marker for the escape key into the key buffer. If the break mechanism is armed then two additional operations take place. Firstly, a special marker is placed into the key buffer that will generate character #EF when it is found (irrespective of the translation tables). This is intended to be used to allow the characters which were in the buffer before the break occurred to be discarded. Secondly, the synchronous break event is 'kicked'.

The break mechanism can be armed or disarmed at any time (by calling KM ARM BREAK or KM DISARM BREAK). The default state is disarmed. When a break is detected the mechanism is disarmed automatically which prevents multiple breaks from occurring.

The method BASIC uses to handle breaks should serve as a model for other programs. BASIC's actions are as follows:

> The break mechanism is armed. After each BASIC instruction the synchronous event queue is polled and if a break event is found (because it has been kicked as explained above) the break event routine is run.
>
> The break event routine stops sound generation (SOUND HOLD) and then it discards all characters typed before the break occurred by reading characters from the keyboard (KM READ CHAR) until either the buffer is empty or the break event marker (character #EF) is found. BASIC then turns the cursor on (TXT CUR ON) and waits for the next character to be typed (KM WAIT CHAR).
>
> If the next character is the escape token (character #FC - the default value generated by the ESC key) then a flag is set to make BASIC abandon execution (or run the user's ON BREAK GOSUB subroutine) and the break event routine returns.
>
> If the next character is any character other than escape then the break will be ignored. If it is any character other than space then this is 'put back' (KM CHAR RETURN). Before the event routine returns the cursor is turned off (TXT CUR OFF), sound generation is restarted (SOUND CONTINUE) and the break mechanism is rearmed. BASIC then continues as if nothing had happened.
>
> When reading or writing from the cassette the ESC key is handled in a different manner which is described in section 8.12.

3.7 Function Keys and Expansion Tokens.

The Key Manager allows for 32 expansion tokens (values #80..#9F) which may be placed in the key translation tables. Each token is associated with a string which is stored in the expansion buffer.

When the user asks the top level for a character a key is fetched from the middle level. If this key is a character it is passed straight back. However, if it is an expansion token then the string associated with the token is looked up. The characters in this string are passed out one at a time with each request for a character until the end of the string is reached. Values #80..#9F and #EF, #FD..#FF in the expansion string are treated as characters and are not expanded or obeyed.

The user may set the string associated with an expansion token (see KM SET EXPAND) and may cause any key on the keyboard to generate an expansion token. The default settings for the expansion tokens and the keys with which they are normally associated are given in Appendix IV. The user may also set the size and location of the expansion buffer (see KM EXP BUFFER); the default buffer is at least 100 bytes long.

3.8 Joysticks.

There may be two joysticks connected to the system. These are both scanned in the same way as keys on the keyboard. Indeed, the second joystick occupies the same locations in the key matrix as certain other keys and is indistinguishable from them. The state of the joysticks can be determined by calling the routine KM GET JOYSTICK.

Because the joysticks are scanned like keys the pressing of joystick buttons can be detected like any other key. Firstly, individual direction or buttons can be tested in the key bit map (see section 3.1) by calling KM TEST KEY. Secondly, the joystick buttons generate characters when they are pressed (providing the translation tables are set suitably) and these characters can be detected. The major problem with this latter method is that the rate of generation of characters depends on how fast the keyboard is set to repeat. If the repeat speed is increased to make the joystick more responsive then the keyboard may become impossible to use.

See Appendix I for the numbering of the keys and joystick buttons and see Appendix II for the default translation tables.

4 The Text VDU.

The Text VDU is a character based screen driver. It controls 8 different streams each of which can have an area of screen allocated to it (a window). The Text VDU allows characters to be written to the screen and read from the screen. It also treats certain 'characters' as 'control codes' which can have various effects, from moving the cursor to setting the colour of an ink.

4.1 Text VDU Coordinate Systems.

The Text VDU uses two coordinate systems - logical and physical. Generally the user specifies positions to the Text VDU in logical coordinates. Physical coordinates are used internally and occasionally by the user to specify positions to the Text VDU. Both systems use signed 8 bit numbers and work in character positions. Each character position is 8 pixels (dots) wide and 8 pixels high. This means that the position of a coordinate on the screen depends upon the screen mode.

Physical coordinates have columns running left to right and rows running top to bottom. The character position at the top left corner of the screen is row 0, column 0.

Logical coordinates are similar to physical coordinates except that the character position at the top left corner of the current text window is row 1, column 1.

4.2 Streams.

The Text VDU has facilities for handling up to 8 streams at once. Each stream has an independent state (although some facilities are shared and thus affect all streams when altered). The features that are stream dependent are:

VDU enable.
Cursor enable (enable or disable, on or off).
Cursor position.
Window size.
Pen and paper inks.
Character write mode (opaque or transparent).
Graphics character write mode.

The features that affect all streams include:

Character matrices.
Control code buffer.
Text VDU indirections.
Screen mode.

All these features are explained in detail in the sections below.

At any time, the stream which is currently selected may be changed without adverse effects provided that the control code buffer is not in use (see section 4.7 for further explanation). A stream will remain selected until another stream is selected. This means that a program need not know which stream it is using.

The default stream, selected at EMS, is stream 0.

BASIC extends the stream concept to include the printer and cassette files. This extension is not part of the firmware.

4.3 Text Pen and Paper Inks.

Each stream has a pen and a paper ink associated with it. The text pen ink is used to set the foreground pixels in characters (see section 4.6). The text paper is used to set the background pixels in characters and to clear the text window.

The pens and papers can be set to any ink that is valid in the current screen mode (see section 6.1). The default setting for a stream has the paper set to ink 0 and the pen set to ink 1. Changing a pen or paper ink does not change the screen; it merely alters how characters will be written in the future.

4.4 Text Windows.

Each stream has a text window associated with it. This window specifies the area of the screen where the stream is permitted to write characters. This allows different streams to use different portions of the screen without interfering with each other.

Windows are trimmed so that they fit within the current screen (whose size varies with the screen mode, see section 6.1). The smallest size window allowed is 1 character wide and 1 character high.

Before writing to the screen the position to write at is forced to lie inside the window (see 4.5 below). This may cause the window to roll. Other operations, such as obeying certain control codes also cause the write position to be forced inside the window.

A text window which does not cover the whole screen is rolled by the firmware copying areas of screen memory around. There is no alternate method available. This makes rolling large windows a fairly time consuming process.

A text window which covers the whole screen is rolled by using the hardware rather than by copying areas of memory. The offset of the start of the screen in the screen memory can be set (see section 6.4). By changing this offset by +80 or −80 the whole screen can be rolled up or down by a line of characters.

It is obviously a good idea to prevent windows that are being used from overlapping. If they are allowed to overlap then the portion in multiple use will merely contain whatever was written to it last. There is no precedence of windows one over another. A window occupying the whole screen will overlap the other windows and so if this window is rolled it will move the contents of the other windows.

The default windows, set up at EMS and after changing screen mode, cover the whole of the screen. All eight windows overlap.

4.5 The Current Position and the Cursor.

Each stream has a current position associated with it. This is where the next character to be printed on the screen is expected to be placed. However, if, when a character is to be printed, the current position is found to lie outside the text window then it is forced inside. The following steps are applied in turn to force the current position inside the window:

1/ If the current position is left of the left edge of the window then it is moved to the right edge of the window and up one line.

2/ If the current position is right of the right edge of the window then it is moved to the left edge of the window and down one line.

3/ If the current position is now above the top line of the window then it is moved to the top line of the window and the contents of the window are rolled down one line.

4/ If the current position is now below the bottom line of the window then it is moved to the bottom line of the window and the contents of the window are rolled up one line.

When the cursor is enabled, the current position is marked by the cursor blob. However, before placing the cursor blob on the screen, the current position is forced to lie inside the current window just as it is before a character is placed on the screen. This may cause the current position to move.

If the cursor is disabled then the current position may lie outside the window and it will not be forced inside the window until, for example, a character is printed.

The current position can be changed directly (by calling TXT SET CURSOR, TXT SET ROW or TXT SET COLUMN) or by sending control codes to the Text VDU. The location the current position is moved to is not forced inside the window immediately, but only when the window is to be written to, as described above. This allows the current position to be changed by moving via a position outside the window, if required.

There are two ways to disable the cursor and prevent the cursor blob from appearing on the screen. The first, cursor on/off, is intended for use by system programs. This is used by BASIC, for example, to hide the cursor unless input is expected. The second, cursor enable/disable, is intended for use by the user. The cursor blob will only be placed on the screen if it is both on and enabled.

The cursor blob is normally an inverse patch. The character at the cursor position is displayed with the text pen and paper inks reversed. This makes it easy to restore the original form of the character position if the cursor is moved. It is possible for the user to alter the form of the cursor blob, if required, by changing the indirections TXT DRAW CURSOR and TXT UNDRAW CURSOR.

4.6 Characters and Matrices.

A character is displayed on the screen in an area 8 pixels (dots on the monitor) wide and 8 pixels high. Thus the maximum number of characters on the screen depends upon the screen mode, (see section 6.1). Each character has a matrix which is an 8 byte vector that specifies the shape of the character. The first byte of the vector refers to the top line of the character and the last byte to the bottom line of the character. The most significant bit of a byte in the vector refers to the leftmost pixel on a line of the character and the least significant bit refers to the rightmost pixel on a line of the character. If a bit in the matrix is set then the pixel is in the foreground. If a bit is clear then the pixel is in the background.

A foreground pixel in the character is always set to the pen ink. The treatment of a background pixel depends on the character write mode of the VDU. In the default mode, opaque mode, background pixels are set to the paper ink. There is another mode, transparent mode, in which the background pixels are not altered. Thus, in transparent mode, the character is written over the top of the current contents of the screen. This is useful for annotating pictures or generating composite characters.

The Text VDU is capable of printing 256 different characters, although special effort is required to print the first 32 characters which are usually interpreted as control codes. The matrices for the characters are normally stored in the ROM but the user may arrange for any number of the characters to have matrices stored in RAM where they may then be altered. The default setting, at EMS, is to have all the matrices in ROM. (BASIC takes special action during its own initialisation to create 16 'user defined' matrices.) The default character set is described in Appendix VI.

When the user sets up a table of user defined matrices, by calling TXT SET M TABLE, it is initialised with the current settings of the matrices from ROM or RAM. This means that extending the table does not alter the current matrices. Contracting the table will make the characters lost revert to their default matrices in ROM.

When characters are read from the screen (by calling TXT RD CHAR) the pixels on the screen are converted to the form of a matrix. This is compared with the current character matrices to find which character it is. This means that changing the character matrices or altering the screen may make a character unrecognisable, in particular, changing the pen or paper ink can cause confusion. Usually these problems result in the character appearing to be a space (character #20) and so special precautions are taken to avoid generating spaces - after some ink changes real spaces may be read as block graphic characters #80 or #8F.

To allow the user to change how characters are written to and read from the screen, the indirections TXT WRITE CHAR and TXT UNWRITE are provided.

4.7 Character Output and Control Codes.

The main character output routine for the Text VDU is TXT OUTPUT. This obeys controls codes (characters 0..31) and prints all other characters. Characters sent to TXT OUTPUT pass through various levels of indirection and can be dealt with by various output routines.

TXT OUTPUT uses the TXT OUT ACTION indirection to sort out whether the character is a printing character, is a control code to be obeyed or is the parameter of a control code.

TXT OUT ACTION normally calls TXT WRITE CHAR to print characters on the screen. However, if the graphic character write mode is selected then characters are printed using the Graphics VDU character write routine (see 5.6 below). This mode can be selected on a character by character basis using a control code or on all characters sent (see TXT SET GRAPHIC). When graphic character write mode is selected control codes are not obeyed but are printed by the graphics routine instead.

TXT OUT ACTION deals with a control code in the following manner:

1/ The code is stored at the start of the control code buffer.

2/ The code is looked up in the control code table to find out how many parameters it requires.

3/ If no parameters are required go directly to step 5.

4/ If one or more parameters are required then TXT OUT ACTION returns but the next characters sent to it are added to the control code buffer rather than being printed or obeyed. This continues until sufficient parameter characters have been received.

5/ The code is looked up in the control code table to get the address of the routine to call to perform the control code and this routine is then executed.

6/ The control code buffer is discarded and the next character sent may be printed or may be the start of a new control code sequence.

The user can change the operation of a control code by changing the entry for it in the control code table (see TXT GET CONTROLS). This contains a 3 byte entry for each code and entries are stored in ascending order (i.e. the entry for #00 first, #01 next and so on).

The first byte of an entry specifies the number of parameters required. This must lie in the range 0..9 as the control code buffer is only capable of storing up to 9 parameters.

The second and third bytes are the address of the routine to call to obey the code. This routine should lie in the central 32K of RAM or in the lower ROM (which will be enabled). It should conform to the following entry/exit conditions:

Entry:

A contains the last character added to the buffer.

B contains the number of characters in the buffer (including the control code).

C contains the same as A.

HL contains the address of the control code buffer (points at the control code).

Exit:

AF, BC, DE and HL corrupt.
All other registers preserved.

The control code buffer is shared between all the streams. A control code sequence should be completed before the stream is changed otherwise unexpected effects may occur.

The default control code actions, set at EMS and when TXT RESET is called, are described in Appendix VII.

It is possible to disable a text stream by calling TXT VDU DISABLE. When disabled the stream will not write any characters to the screen. Normal operation can be restored by calling TXT VDU ENABLE. Note, however, that calling these routines will empty the control code buffer. This effect may be used to avoid problems when the state of the control buffer is unknown (when printing an error message perhaps).

5 The Graphics VDU.

The Graphics VDU allows individual pixels (dots) on the screen to be set or tested and lines to be drawn. The plotting takes place on an ideal screen that is always 640 points wide and 400 points high. This means that more than one point on the ideal screen will map onto a particular pixel on the real screen. The width of the ideal screen (640 points) is chosen to be the horizontal number of pixels on the screen in the highest resolution mode (mode 2). The height of the ideal screen (400 points) is chosen to be twice the vertical number of pixels on the screen in all modes. This ensures that the aspect ratio of the screen is approximately unity, i.e. a circle looks circular and not elliptical.

5.1 Graphics VDU Coordinate Systems.

The Graphics VDU uses 4 coordinate systems. The user specifies positions in user coordinates or relative coordinates or occasionally in standard coordinates. Internally the Graphics VDU uses base coordinates (or occasionally standard coordinates).

User coordinates, relative coordinates and standard coordinates are all very similar. They all use signed 16 bit numbers and work in points with X-coordinates running left to right and Y-coordinates running bottom to top. The screen is always 400 points high and 640 points wide whatever the screen mode is. This means that a pixel (dot on the screen) is mapped onto by 8 points in mode 0, 4 points in mode 1 and 2 points in mode 2. The origin (coordinate (0,0)) of these systems vary:

> In standard coordinates the origin is the point at the bottom left corner of the screen.
>
> The origin of user coordinates can be set by the user. The default origin is at the bottom left corner of the screen. This makes the default user coordinates the same as standard coordinates.
>
> The origin of relative coordinates is the current position (see 5.2 below). This allows plotting to be carried out independently of the position on the screen and is useful if a particular shape is to be repeated on the screen a number of times or if it is inconvenient to keep track of the current location.

Base coordinates are a physical coordinate system which deals with pixels. X-coordinates run left to right and Y-coordinates run bottom to top. Pixel (0,0) is the pixel at the bottom left corner of the screen. Because this coordinate system works in pixels the coordinates of positions on the screen depend upon the screen mode. Base coordinates are unsigned 16 bit numbers and only coordinates that refer to a pixel on the screen are valid.

Graphics routines convert from relative coordinates to user coordinates, if necessary, and then from user coordinates to base coordinates before accessing the physical screen. During the latter conversion there is a loss of accuracy because of the mapping of multiple points onto a single pixel. This could make shapes drawn on the screen appear asymmetrical (particularly circles) but the Graphics VDU avoids this by rounding the coordinates towards the user origin. Thus symmetrical shapes should be drawn symmetrically about the user origin to take advantage of the rounding. If the shape is not centred on the user origin then the asymmetry may reappear.

5.2 The Current Graphics Position.

The Graphics VDU stores a current position. This is the user coordinate of the last point specified to the Graphics VDU (or the origin after clearing the graphics window). The origin of relative coordinates is specified to be at this point, thus relative coordinates are an offset from the current position.

When drawing a line one end is at the position specified while the other end is at the current graphics position. When drawing a character on the screen using the graphics character write routine the character is placed with the current graphics position being the top left corner of the character.

After plotting or testing a point or drawing a line the current graphics position is moved to the position specified. After writing a character the current graphics position is moved right by the width of a character ready for the next character to be written.

5.3 Graphics Pen and Paper Inks.

The Graphics VDU has a pen (foreground) ink and a paper (background) ink. The graphics pen ink is used to plot pixels, to draw lines and to set foreground pixels when writing characters using the graphics write routine (see 5.6 below). The graphics paper ink is used to clear the graphics window and to set background pixels when writing characters using the graphics write routine.

The pen and paper can be set to any ink valid in the current screen mode (see section 6.2). The default has the paper set to ink 0 and the pen set to ink 1. Changing the pen or paper ink does not change the screen it merely alters how pixels will be written in the future.

5.4 Graphics Write Mode.

Pixels plotted by the Graphics VDU are plotted using the current graphics write mode. This specifies how the ink to be plotted interacts with the ink a pixel is currently set to.

There are four write modes:

0: FORCE:	NEW = INK
1: EXCLUSIVE-OR:	NEW = INK xor OLD
2: AND:	NEW = INK and OLD
3: OR:	NEW = INK or OLD

NEW is the ink that the pixel will be set to.
OLD is the ink that the pixel is currently set to.
INK is the ink that is to be plotted.

The default Graphics write mode is FORCE mode. The Text VDU always sets pixels as if it is operating in this mode. Also the graphics window is cleared by writing in FORCE mode irrespective of the actual write mode.

Provided that suitable ink settings are chosen, AND mode and OR mode allow particular bits in a pixel to be cleared or set. This allows the Graphics VDU to write in 'bit planes' and by choosing the colours of the inks carefully overlapping shapes can be drawn and automatically hidden behind one another.

If the inks are chosen suitably, EXCLUSIVE-OR mode can be used to plot over the current contents of the screen. It is also useful because a shape can be removed from the screen by redrawing it since exclusive-oring with the same ink twice restores the original setting of a pixel.

The graphics write mode may be set by calling SCR ACCESS or by using a control code (see Appendix VII).

5.5 Graphics Window.

The Graphics VDU allows a single window to be specified. This allows the user to mix text and graphics on the screen without them interfering with each other. If the text windows are allowed to overlap the graphics window then the contents of the graphics window will be moved when the text windows are rolled. The graphics window cannot be rolled.

When plotting points, drawing lines or writing characters no pixel outside the graphics window is ever written. Unlike the text windows no action is taken to force a point inside the window - actions outside the window will be lost. Conversely, when testing points, points outside the window are all deemed to be set to the current graphics paper ink. Points inside the window are written and read as expected.

The graphics window is related to a specific area of the screen and not to the user coordinate system. Thus, changing the origin of the user coordinate system will not move the location of the window on the screen although it does change the user coordinates of points in the window.

The default graphics window, set at EMS and after changing screen mode, covers the whole of the screen.

5.6 Writing Characters.

The Graphics VDU write character routine draws a character with the current graphics position at the top left corner of the character. The current position is moved right by the width of a character in the current screen mode. The distance moved varies; in mode 0 it is 32 points; in mode 1, 16 points; and in mode 2, 8 points. Control codes, characters 0..31, are printed and are not obeyed.

The character is always written opaquely irrespective of what mode the Text VDU is using to write characters. i.e. The character background is set to the graphics paper ink and the foreground is set to the graphics pen ink. However, the current graphics write mode is used to plot the pixels in the character (see 5.4 above).

6 The Screen Pack.

The Screen Pack is used by the Text and Graphics VDUs to access the hardware of the screen. It also controls the features of the screen that affect both the Text VDU and Graphics VDU, such as what mode the screen is in.

6.1 Screen Modes.

The screen has three modes of operation, numbered 0, 1 and 2. The modes have different resolutions and display different numbers of inks on the screen.

All modes have a vertical resolution of 200 pixels (picture elements or dots on the screen). The horizontal resolution varies from 160 pixels to 640 pixels. As each character is 8 pixels by 8 pixels the number of characters across the screen varies with the mode - from 20 characters to 80 characters. The screen is always 25 characters high.

The number of inks that can be displayed on the screen varies with the screen resolution. When the screen is 640 pixels wide only 2 inks can be displayed, when the screen is 320 pixels wide 4 inks can be displayed and when the screen is 160 pixels wide 16 inks can be displayed.

In summary, the modes are:

Mode	Pixel Size	Character Size	Inks
0	160 x 200	20 x 25	16
1	320 x 200	40 x 25	4
2	640 x 200	80 x 25	2

The default screen mode, set at EMS, is mode 1.

The screen mode is set by calling SCR SET MODE which also has other effects.

Firstly, the screen is cleared to ink 0. If the text and graphics paper inks are not set to ink 0 then this will become apparent on the screen when characters are written or windows are cleared. If the user wishes to alter this screen clearing operation for some reason then it may be intercepted at the SCR MODE CLEAR indirection.

Secondly, the Text and Graphics VDUs are set into standard states. The windows are all set to cover the whole screen. If the pen and paper inks are out of range for the new mode then they are masked (with #01 or #03) to bring them into range. The current text positions are moved to the top left corner of the screen and the text cursors are turned off (see TXT CUR OFF). The current graphics position and the user origin are moved to the bottom left corner of the screen.

6.2 Inks and Colours.

The various screen modes allow pixels (dots on the screen) to be set to different numbers of inks as follows:

Mode 0:	16 inks, 0..15
Mode 1:	4 inks, 0..3
Mode 2:	2 inks, 0..1

How the ink for a pixel is encoded into a byte of screen memory is described in section 6.4. The ink that a pixel is set to determines what colour the pixel is displayed in. However, the colour associated with an ink is not fixed, it can be changed.

There are 27 colours available. Each ink may be set to any of these colours. The border to the screen acts much like an ink and can have its colour specified as well. The display hardware fetches the ink value from the screen memory for each pixel as it is displayed. This ink value is used to access a small area of RAM inside the gate array called the 'palette'. The palette contains the actual colour which is to be displayed by the monitor for that particular ink. Changing entries in the palette thus causes all pixels set to that ink to change colour when they are next displayed (i.e. within 1/50th of a second or so).

In fact the Screen Pack allows two colours to be associated with an ink (or the border). These are loaded into the palette alternately under software control. If the two colours associated with an ink are different then the ink will flash, if the colours are the same then the ink will be steady. The user can change the rate of alternation, from the default of 5 cycles per second, if required (see SCR SET FLASHING).

When specifying colours the Screen Pack uses an ordering that corresponds to a grey scale on a monochrome monitor. This runs from the darkest colour (black), colour 0, to the brightest colour (bright white), colour 26. The colours do not appear to have any particular ordering when viewed on a colour monitor.

The palette uses a different (and apparently nonsensical) numbering scheme for the colours. The Screen Pack automatically translates the grey scale number to the hardware number and vice versa when appropriate. Unless the user is driving the hardware directly the hardware numbers will never be encountered.

The default settings for the colour of each ink and a list of the 27 colours available are given in Appendix V.

6.3 Screen Addresses.

The Screen Pack does not use a coordinate system itself. It uses screen addresses. However, it does work with the physical and base coordinate systems of the Text and Graphics VDUs described in sections 4.1 and 5.1 respectively. In particular, routines are provided to convert positions given in physical or base coordinates to screen addresses.

A screen address is, prosaically enough, the address of a byte within the screen memory. To specify a particular pixel a screen address is often passed to a routine along with a mask that indicates exactly which pixel is required. Routines are provided for stepping a screen address up, down, right and left one byte. (The screen map makes this a non-trivial operation.)

6.4 Screen Memory Map.

The screen is a memory mapped pixel screen. The screen memory fills 16K of RAM in all modes. The default location for the screen, set at EMS, is the 16K of RAM starting at #C000. This lies underneath the upper ROM, when it is enabled, which keeps the screen out of the way of the rest of the system. However, this also means that the upper ROM has to be disabled whenever the screen is read. The firmware jumpblock uses LOW JUMP restarts which turn the upper ROM off to ensure that the screen memory is accessible if required.

It is possible to change the location of the screen memory to any of the 4 16K memory blocks on 16K boundaries (see SCR SET BASE). However, only #C000 and #4000 are useful; #0000 and #8000 both overlap firmware jumpblocks or other system areas. The descriptions below all assume the default screen location at #C000.

The screen memory map is not simple. Fortunately it is not necessary to understand it because the Text and Graphics VDUs provide idealised models of the screen. However, to achieve maximum speed for certain applications (such as animated games) it may be necessary to access the screen memory directly.

The screen memory is divided into 8 blocks, each 2K bytes long. Block 0 runs from #C000 to #C7FF, block 1 runs from #C800 to #CFFF, and so on. Each line of pixels on the screen uses 80 consecutive bytes from a block. The top line of the screen comes from block 0, the second line from block 1 and so on until the eighth line which comes from block 7. The sequence starts with block 0 again on the ninth line and repeats in this fashion all the way down the screen. The successive lines in a block are stored consecutively so there are 48 unused bytes at the end of each block.

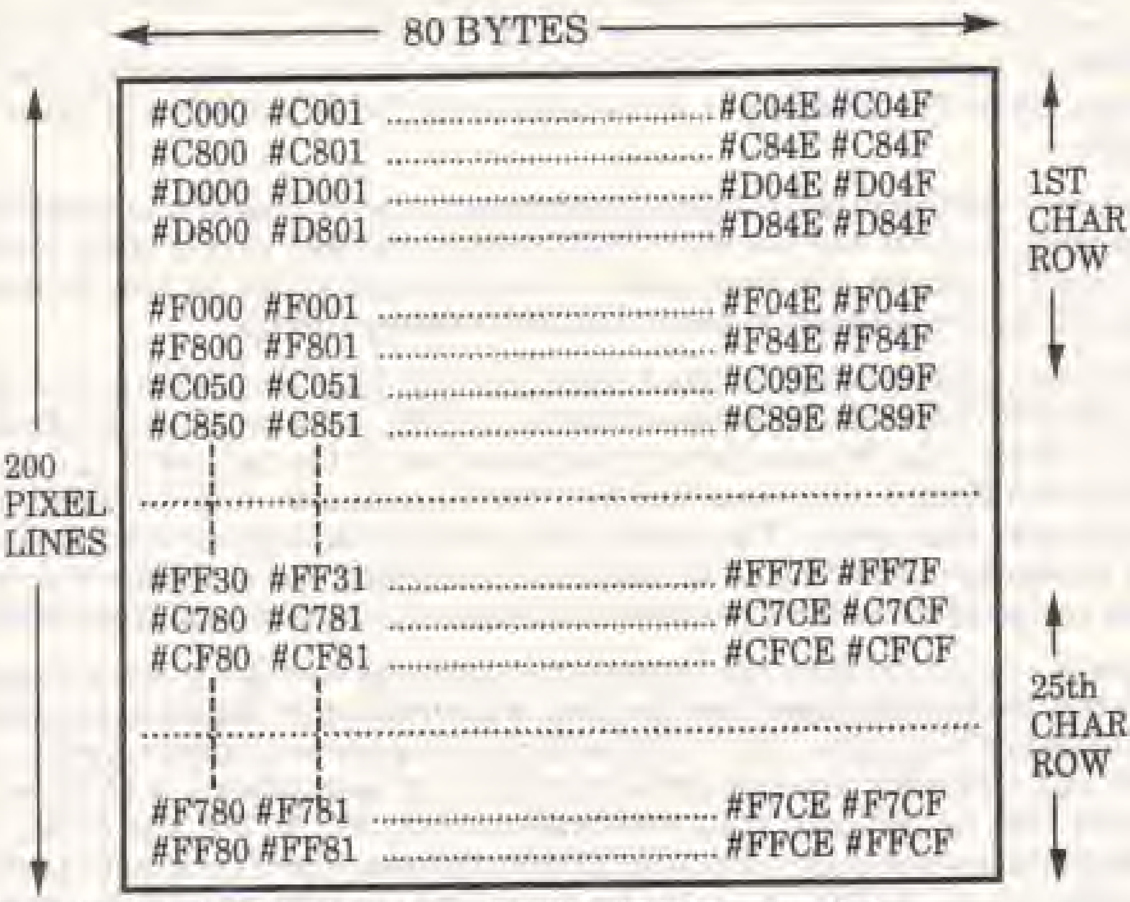
There is a further complication to this screen map. The description above assumes that the first byte displayed from the block is the first byte of the block. In practice the offset in a block of the first byte to be displayed can be set to any even value (see SCR SET OFFSET). The same offset applies to all eight blocks. A block wraps around from its last byte to its first byte, thus #C7FE, #C7FF and #C000 are consecutive bytes in block 0 and could all be on the same line of the screen. Altering the offset by +|− 80 MOD 2048 (the length of a line) rolls the screen up or down by one character line (8 pixel lines). This effect is used by the Text VDU when rolling the entire screen.

The meaning of the bytes accessed as described above varies with the screen mode. Each byte stores the inks for 2, 4 or 8 pixels. The bits used to encode each pixel are not arranged in an obvious manner. The following table specifies which bits of screen memory are used to encode which pixel in the various modes. The bit numbers given in the table are the bits of the screen byte. They are given in the

order of bits in the pixel - the first bit given is most significant bit of the pixel and the last bit is the least significant bit.

	Mode 0	Mode 1	Mode 2
Leftmost pixel	Bits 1,5,3,7	Bits 3,7	Bit 7
.			Bit 6
.		Bits 2,6	Bit 5
.			Bit 4
.	Bits 0,4,2,6	Bits 1,5	Bit 3
.			Bit 2
Rightmost pixel		Bits 0,4	Bit 1
			Bit 0

The following diagram illustrates the mapping from pixels on the screen to addresses in screen memory for the simple case of a base address of #C000 and an offset of 0.



#C7D0..#C7FF, #CFD0..#CFFF, ... , #FFD0..#FFFF are unused.

7 The Sound Manager.

The Sound Manager deals with the sound chip. It allows various envelopes and sounds to be set up and played under the control of the user. Most of the control is achieved using software rather than the sound chip hardware.

7.1 The Sound Chip.

The sound chip used is the General Instruments AY-3-8912. This has three channels and a pseudo-random noise generator that can be connected to any of the channels. The chip has a limited number of amplitude envelopes available (see Appendix IX) but the software enveloping, described below, can achieve all that the hardware is capable of, and more. Tone enveloping is all done by the software, there is no hardware support.

The sound generated by the chip uses square waveforms. There is no way to generate any other waveform.

It is possible to access the sound chip directly should the need arise. However, the routine MC SOUND REGISTER should be used to write to registers of the sound chip. This is because the keyboard is attached to the I/O port of the sound chip and the keyboard scanning routine expects to find the sound chip in a standard state (i.e. not in use). Also, there are timing constraints on accesses to the chip; using MC SOUND REGISTER will avoid consideration of these details.

The sound chip has three independent sound channels. The outputs from these are mixed together to form two stereo channels - sound channels A and B are mixed to form one stereo channel and sound channels B and C are mixed to form the other stereo channel. The stereo sound is available on the output jack on the back of the machine. However, there is only a single internal speaker and so the two stereo channels are mixed together to drive this. The volume of sound from the internal speaker can be controlled by the volume control knob on the side of the machine near the on/off switch. This control overrides the other volume control methods described below.

7.2 Tone Periods and Amplitudes.

The sound chip allows 16 different amplitudes in the range 0..15. Amplitude 0 is no sound at all, amplitude 15 is maximum volume.

The pitch of a note to be generated is specified by the period of the note rather than by the frequency. This period is given in 8 microsecond units. Thus, the tone period specified and the frequency of the tone generated are related by the formula:

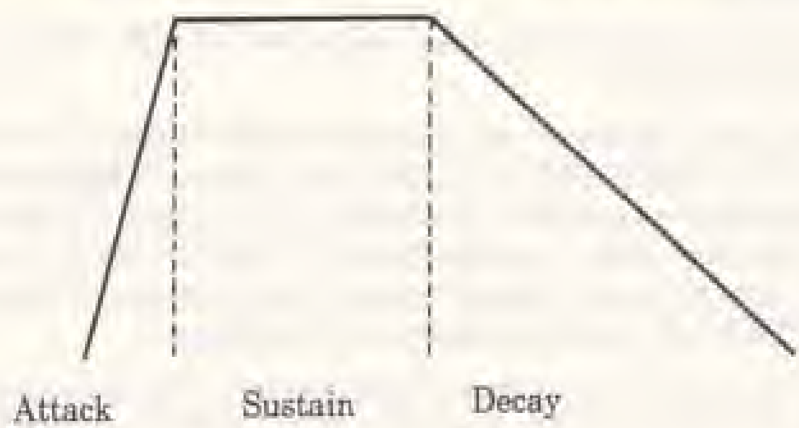
Tone period = 125 000 / Frequency

See Appendix VIII for a list of the suggested periods for generating musical notes.

7.3 Enveloping.

Real sounds rarely have a constant volume. Enveloping allows an approximation to the variation in volume of real sounds to be made. The sound is split into a number of sections each of which can increase the volume, decrease the volume, or keep it constant. The length of these sections can be varied, as can the rate of increase or decrease in volume. For example, a note generated by a musical instrument may be considered to have 3 sections as follows:

Attack:	The volume of the note rises rapidly to its peak.
Sustain:	The volume of the note remains constant while the note is played.
Decay:	The volume falls away slowly to zero as the note finishes.



The Sound Manager allows two types of envelopes; amplitude envelopes to control a sound's volume and tone envelopes to control its pitch (the pitch is varied in much the same way as the volume). The user can set up to 15 different envelopes of each type. The exact formats of the data blocks specifying envelopes are given in SOUND AMPL ENVELOPE and SOUND TONE ENVELOPE.

a. Amplitude envelopes.

An amplitude envelope is used to control the volume and length of a sound. It can have up to five sections. Each section can be either a hardware or a software section. Software sections are either absolute or relative.

Hardware sections write values into the sound chip registers 11, 12 and 13 to set up a hardware envelope. (See Appendix IX for a description of the sound chip registers). Generally a hardware section will be followed by a software section that does nothing except wait for a time long enough for the hardware envelope to operate.

An absolute software section specifies a volume to set and a time to wait before obeying the next section.

A relative software section specifies an step size, a number of steps and a time to wait. For each step requested, the current volume is changed by the given step size and then the Sound Manager waits for the given time after each step before obeying the next step.

Amplitude envelopes are set by calling SOUND AMPL ENVELOPE.

b. Tone envelopes.

A tone envelope controls the pitch of the sound. It can have up to five sections. Each section can be either an absolute or a relative section. The sections of a tone envelope are not necessarily related to those of an amplitude envelope.

An absolute section specifies a tone period to set and a time to wait before obeying the next section.

A relative section specifies an step size, a number of steps and a time to wait. For each step requested, the current tone period is changed by the given step size and then the Sound Manager waits for the given time after each step before obeying the next step.

If the tone envelope is completed before the sound duration expires (see section 7.4f) then the final pitch is held constant. Alternatively, tone envelopes can be set to repeat themselves automatically. This allows tremulo effects to be created.

Tone envelopes are set by calling SOUND TONE ENVELOPE.

7.4 Sound Commands.

When a sound is given to the Sound Manager to be played, by calling SOUND QUEUE, a lot of information needs to be specified. This is described briefly below. The detailed layout of a sound command data block is described in Appendix X.

a. Initial tone period.

The sound is issued with an initial tone period. The pitch of the sound can be varied from this initial value using a tone envelope. If no tone envelope is specified the pitch remains constant. An initial tone period of zero means no tone is to be generated, presumably the sound is to be pure noise (see (e) below).

b. Initial volume.

The sound is issued with an initial volume. The volume of the sound can be varied from this initial value using an amplitude envelope. If no amplitude envelope is specified then the volume remains constant.

c. Tone envelope.

This specifies which tone envelope to use. If no envelope is specified then the pitch of the sound remains constant.

d. Amplitude envelope.

This specifies which amplitude envelope to use. If no envelope is specified then default system envelope is used. This keeps the volume of the sound constant and lasts for 2 seconds.

e. Noise period.

If the noise period is zero then no noise is to be added to the sound. Any other value sets the period for the pseudo-random noise generator and adds noise to the tone generated. Note that there is only one noise generator and so if two sounds are to use it at the same time they will need to agree on the period.

f. Duration.

The length of a sound can be specified in two ways, either as an absolute time (duration) or as a number of operations of the amplitude envelope. In the latter case the envelope is run one or more times and the sound finishes when the envelope has been executed the specified number of times. In the former case, if the duration finishes before the envelope (if any) then the sound is cut short. If the duration is longer than the envelope then the final amplitude is held until the duration expires.

g. Channels and Synchronisation Bits.

The sound can be issued to one or more channels. If a sound is issued to more than one channel then these channels automatically rendezvous with each other. Rendezvous requirements can be set explicitly as well. Also the sound can be held or the sound queue can be flushed (see section 7.6).

7.5 Sound Queues.

Each channel has a queue associated with it. Each queue has space to store at least three sounds. The sound at the head of each queue may be running and making music on its channel or it may be waiting for various synchronisation requirements (see 7.6 below). When a sound command is issued the sound is placed into the queues for the channels specified by the command. When the sound reaches the head of the queue, and providing its synchronisation requirements are met, it is executed.

If a sound that has the flush bit set is put into a queue then all sounds queued for that channel are discarded and any executing sound is stopped immediately. Thus a sound with the flush bit set will move to the head of the queue immediately and may commence execution.

A routine (SOUND CHECK) is provided to test the status of the sound at the head of a queue and to determine how much free space is in a queue. It is also possible to set up a sound event for each queue (by calling SOUND ARM EVENT). This synchronous event is 'kicked' when the queue has a free space in it. The sound event mechanism allows the generation of sound to be carried on as a background task whilst some other action is being carried out.

7.6 Synchronisation.

There are two mechanisms to allow sounds on different channels to be synchronised. These are holding sounds and rendezvous. The purpose of synchronisation is to ensure that sounds start simultaneously. For example, a simulation of an instrument might use one channel to generate the fundamental note and another channel to generate the harmonics of the note. The synchronisation mechanism, particularly rendezvous, may be used to ensure that the fundamental and the harmonic sounds start exactly together.

A sound can be specified to be held when it is issued. This means that when it reaches the head of the sound queue it is not executed immediately. Instead it waits until it is explicitly released (by calling SOUND RELEASE) before it starts execution.

A sound can have rendezvous requirements set on it when it is issued. If a sound is issued to more than one channel then these channels all set rendezvous with each other automatically. When a sound with a rendezvous set reaches the head of the sound queue it is not executed immediately. Instead it waits until sounds with matching rendezvous requirements reach the head of their sound queues. Only when all rendezvous sounds are found to be present and ready to run do they start.

For instance, a sound on channel A marked to rendezvous with a sound on channel B will not start until a sound on channel B marked to rendezvous with channel A is ready to start - and vice versa! If a sound is ready to start on channel B that is not marked to rendezvous with channel A then it starts but the sound on channel A continues to wait for its rendezvous.

7.7 Holding Sounds.

It is possible to stop a sound while it is executing by calling SOUND HOLD. This will stop a channel making any sound and will save the state of the sound. The sound can be restarted from where it was held by calling SOUND CONTINUE. However, if a hardware envelope was running when the sound was held then it is impossible to predict the effect of restarting the sound. The hardware envelope may or may not continue from where it was held.

Calling SOUND HOLD is different from setting the hold bit when issuing a sound as described in section 7.6 above. SOUND HOLD stops all sounds being generated at any time whilst the hold bit is a method for synchronising sounds and prevents a particular sound starting when it reaches the head of the queue.

8 The Cassette Manager.

The Cassette Manager deals with reading files from and writing files to tape. These operations can either be performed on a character by character basis or on a whole mechanism file at once. The built in cassette mechanism ('Datacorder') is completely controlled by software. There is no hardware support for the cassette, even the timing for reading and writing bits is performed by software.

The format of data on the tape is described in great detail. This will only be of academic interest to most users. More general information can be found in sections 8.4 onwards.

8.1 File Format.

A file on tape is split into blocks each with a header record and a data record containing up to 2K (2048) bytes of data. The cassette motor which is under software control is turned off between each file block to allow time to process the data read or to generate the data to be written. The motor start-up gap also serves to separate the blocks from each other.

The general format of a block is as follows:

Motor Start-up	File header record	File data record

However, the first and last blocks of a file have an extra pause before and after them respectively, to separate files on the tape. Their formats are thus:

First block:

Motor start-up	Pre-file gap	File header record	File data record

Last block:

Motor start-up	File header record	File data record	Post-file gap

There is a strong distinction between the file header record and the file data record. The header record is written using one synchronisation character (#2C) and the data record with another (#16). This means that when the Cassette Manager is searching for a file header it is impossible for it to find a file data record by mistake, and vice versa. See 8.2 below for the use of the synchronisation characters.

8.2 Record Format.

A record can contain any number of data bytes from 1 to 65536. The data is split into segments each of which is 256 bytes long. The last segment is padded out to 256 bytes with zeros when writing if necessary. When reading a record any extra bytes are ignored although they are accumulated into the CRC.

The layout of a record is as follows:

Leader	Segment 1	....	Segment N	Trailer

There are N segments where 256*N is the length of data (plus padding) to be written.

A file header record always contains one segment; a file data record contains from one to eight segments (usually 8 segments).

a. Leader

At the start of all records a leader is written which has the following layout:

Pre-record gap	2048 one bits	Zero bit	Sync byte

The leading gap is there to ensure the failure of any attempt to synchronise on the end of a preceding record or on data that was on the tape and that has been over-recorded.

The long sequence of one bits is used to calculate the speed at which the data was written and hence to calculate the threshold value used to distinguish one bits from zero bits.

The single zero bit is used to mark the impending end of the leader and is also used to determine whether the recording has been inverted (see section 8.3).

The synchronisation byte is there to help prevent spurious synchronisation on sequences of bits such as might be found in a record. If an incorrect value for the sync byte is found then an attempt has been made to synchronise on the middle of a record or on the wrong type of record. This byte is used to distinguish header records from data records in a file block (header records use #2C while data records use #16)

b. Segments

Each segment contains 256 data bytes and has the following format:

Byte 1	Byte 2	....	Byte 256	CRC 1	CRC 2

'CRC 1' is the more significant byte and 'CRC 2' the less significant byte of the logical NOT of the CRC calculated for the 256 bytes in the segment. (The CRC polynomial used is '$X^{16}+X^{12}+X^{5}+1$' with an initial seed of #FFFF).

c. Trailer

The trailer is simply an extra 32 one bits written to the end of the record.

8.3 Bit Format.

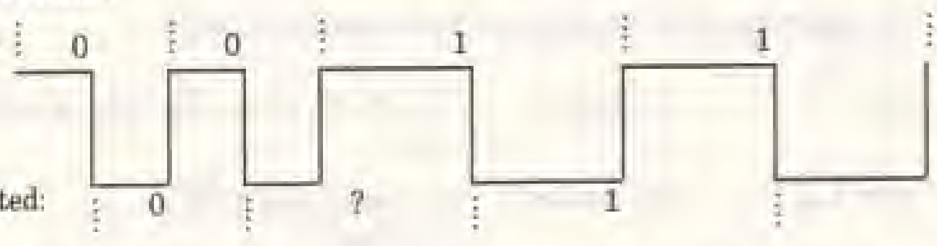
A bit is written to the tape as a period of low level followed by an equal period of high level. A one is written to the tape with these periods twice as long as those for a zero. The length of the period for a zero can be set by the user (see CAS SET SPEED).

The tape circuitry has a tendency to move the positions of edges (transitions from high to low or low to high) so as to balance out the difference between ones and zeros written to tape. Precompensation is used - which adds to the period of a one bit and subtracts from the period of a zero bit to make the waveform closer to the ideal when it is read.

When reading, the speed at which the recording was made is determined by timing the one bits in the record leader. As this is a long sequence of the same bit the edges are not shifted and no precompensation is applied. Since the speed is established independently for each record this automatically takes into account most tape speed variations.

Data is written low-high but may be inverted when read (i.e. high-low). It is important for the firmware to determine whether the waveform being read is inverted or not. If this is not achieved then the bits will not be read properly as the following example shows:

Inversion detected:



Inversion not detected:

The zero bit in the record leader is used to determine whether the recording has been inverted.

Bytes written to the tape are written with the most significant bit first and the least significant bit last.

8.4 The Header Record.

The header record in a file block contains information about the file and about the data in the following data record. Some of the entries in the header are used by the system for various purposes. The remaining entries are available for the user to set when writing a file, and to read when reading a file. These entries are the file type (byte 18) and all the user fields (bytes 24..63) including the logical length (bytes 24..25) and the entry address (bytes 26..27). The user fields will all be set to zero if they are not used.

The header is laid out as follows:

System fields

Bytes 0..15	Filename	Padded to 16 bytes with nulls.
Byte 16	Block number	The first block is normally block 1 and block numbers increase by 1 on successive blocks.
Byte 17	Last block	A non-zero value means that this is the last block of a file.
Byte 18	File type	A value recording the type of the file (see below).
Bytes 19..20	Data length	The number of data bytes in the data record.
Bytes 21..22	Data location	Where the data was written from originally.
Byte 23	First block	A non-zero value means that this is the first block of a file.

User fields

Bytes 24..25	Logical length	This is the total length of the file in bytes.
Bytes 26..27	Entry address	The execution address for machine code programs.
Bytes 28..63	Unallocated	These are unallocated and may be used as required.

The file type (byte 18) is split into a number of fields:

Bit 0	Protection	If this bit is set the file is protected in some way.
Bits 1..3	File contents	0 = Internal BASIC. 1 = Binary. 2 = Screen image. 3 = ASCII. 4..7 are unallocated.
Bits 4..7	Version	ASCII files should be version 1, all other files should be version 0.

8.5 Read and Write Speeds.

The Cassette Manager is capable of reading and writing data at speeds ranging from 700 baud to 2500 baud. There are two speeds commonly used in this range, 1000 baud (the default speed selected at EMS) and 2000 baud. The default speed is chosen to be near the slowest speed to give maximum reliability. The reliability at 2000 baud is still good, however, particularly when playing back on the same machine that was used to record a tape.

Bits are written to the tape as a single cycle of a tone. The tone for a one always has half the frequency of the tone for a zero. Thus ones are twice as long as zeros on the tape. This means that the baud rates given above are only averages and vary according to the actual data written.

Even with the built in cassette mechanism the Cassette Manager has to precompensate the waveform written to the tape to achieve the speeds quoted. This means that the lengths of bits written are altered (ones lengthened, zeros shortened) to try to make the waveform read closer to the ideal after the edges of the waveform have been shifted by the cassette circuitry.

It is only necessary to set the Cassette Manager's write speed. When reading a record from tape the record leader is used to calculate the speed at which it was written. This also allows for tape speed variations between different machines.

8.6 Cataloguing.

To generate a catalogue from the tape the Cassette Manager reads a sequence of file blocks and prints information from them. The file blocks may come from any file, in any order. Cataloguing continues until the user hits the escape key.

The information is reported as follows:

```
FILENAME   block N L Ok
```

FILENAME is either the name of the file or 'Unnamed file' if the filename starts with a null.

The block number, N, indicates which block of the file it is. Normally block 1 is the first block of a file.

L is a character representing the file type and protection status of the file. It is formed by adding #24 (character '$') to the file type from the header masked with #0F. This gives the following characters:

$	an unprotected BASIC program.
%	a protected BASIC program.
&	a binary file.
'	a protected binary file
*	an ASCII file.

Other characters are possible but the above are the standard file types that are written by the on-board ROM.

The above information is printed when the header record is read correctly.

Ok is printed after the data record has been read correctly.

8.7 Reading Files.

Before a file can be read from it must be opened (by calling CAS IN OPEN). This sets up the filename (see 8.10 below) and reads the first block of the file so that the header can be inspected.

The file may either be opened for character input or for direct input, but not both. The mode of input is set by the first access to the file and not when it is opened. As soon as one mode is selected it becomes impossible to use the other mode to access the file.

Character input (calling CAS IN CHAR) allows the user to read the file sequentially one character at a time. Blocks of the file are read from tape into the buffer as needed. This is intended for reading text files and similar applications.

Direct input (calling CAS IN DIRECT) reads the whole of the file into memory in one go. This is intended for loading machine code programs or screen dumps and similar applications.

Interrupts are disabled whilst reading from tape because this has serious timing constraints. Disabling interrupts will prevent the various timer interrupts (as described in section 10.1) from occuring. In particular this might leave the sound chip making a noise for a long period of time and so the Sound Manager is shut down (see SOUND RESET).

8.8 Writing Files.

Before a file can be written to it must be opened (by calling CAS OUT OPEN). This sets up the filename (see 8.10 below) and the rest of the header that will be written in each file block.

The file may either be opened for character output or for direct output, but not both. The mode of output is set by the first write to the file and not when it is opened. As soon as one mode is selected it becomes impossible to use the other mode to write to the file.

Character output (calling CAS OUT CHAR) allows the user to write to the file one character at a time. The characters are buffered until a complete block (2048 characters) is ready to be written whereupon a file block is written to the tape.

Direct output (calling CAS OUT DIRECT) writes the whole of the file from memory in one go. The data written is still packaged into 2048 byte blocks.

Whichever output mode is used, it is important to close the output file properly (using CAS OUT CLOSE) otherwise the last block of the file will not be written.

Interrupts are disabled whilst writing to tape because this has serious timing constraints. Disabling interrupts will prevent the various timer interrupts (as described in section 10.1) from occuring. In particular this might leave the sound chip making a noise for a long period of time and so the Sound Manager is shut down (see SOUND RESET).

8.9 Reading and Writing Simultaneously.

The Cassette Manager allows two files to be open simultaneously. One must be open for reading and the other for writing. Thus it is possible to read from one file and write to another file at the same time.

When the Cassette Manager is about to read a block it asks the user to press PLAY and this implies that the tape with the file for reading should be loaded. Similarly, when it is about to write a block it asks the user to press REC and PLAY and this implies that the tape to which the file is to be written should be loaded. The Cassette Manager assumes that the tape is not changed and that the appropriate cassette controls remain pressed as requested until a prompt is issued. It also assumes that pressing a key means that the prompt has been obeyed.

It is unwise to attempt to read and write simultaneously with the Cassette Manager messages turned off. The only notification given of which tape should be loaded is in the prompt messages.

8.10 Filenames.

When the user opens a file for reading or writing the name of the file to be read or written is specified. The filename is a string of any 16 characters (#00..#FF). If the file name specified is longer than 16 characters then it is truncated and if it is shorter than 16 characters it is padded to 16 characters with nulls (character #00).

When opening a file for reading a zero length filename or one that starts with a null has a special meaning - read the next file on the tape. The Cassette Manager searches the tape until it finds the first block of a file and it reads this file. Once the first block of a file has been found the Cassette Manager will only read from that file and no other.

BASIC uses a slightly extended form of the filename. If the first character of a BASIC filename is an exclamation mark (character #21) the BASIC turns the prompt messages off (see 8.11 below) and removes the exclamation mark from the name. This facility is not provided at the Cassette Manager level.

8.11 Cassette Manager Messages.

The Cassette Manager issues a number of messages to prompt and inform the user and to warn when errors have occurred. The messages that prompt or inform the user may be turned on or off as desired (see CAS NOISY). Messages that inform the user of errors cannot be turned off by this mechanism.

a. **Prompt messages.**

```
Press PLAY then any key:
```

This message is issued when the Cassette Manager is about to read the first block of a file from tape or when it is about to read a block after having written to tape (see section 8.9). It indicates that the tape containing the file to be read should be loaded and that the PLAY button on the recorder should be pressed. The Cassette Manager does not issue this message at other times since it assumes that the correct tape is still loaded and that the PLAY button is still pressed.

```
Press REC and PLAY then any key:
```

This message is issued when the Cassette Manager is about to write the first block of a file to tape or when it is about to write a block after having read from tape. It indicates that the tape on which the file is to be written should be loaded and that the REC and PLAY buttons on the recorder should be pressed. The Cassette Manager does not issue this message at other times since it assumes that the correct tape is still loaded and that the REC and PLAY buttons are still pressed.

b. **Information messages.**

```
Found FILENAME block N
```

This message is printed when reading from the tape if a header record is found that for any reason does not match the record that was expected. This may indicate that the tape is positioned incorrectly (too early or too late) or that the wrong tape is being played.

```
Loading FILENAME block N
```

A block of the file has been found and is being read from tape.

```
Saving FILENAME block N
```

A block of the file is being written to tape.

FILENAME in the above messages is the name of the file or 'Unnamed file' if the filename starts with a null.

The block number, N, indicates which block of the file is being read or written. The first block of a file is normally block 1, the second block 2 etc.

c. **Error messages.**

```
Rewind tape
```

While searching for a block of the file being read, a higher numbered block than that required has been found. The required block has been missed. This message is often produced after a read error in the required block when the next block is found.

```
Read error X
```

An error of some kind occurred whilst reading from the tape. The tape should be rewound and the block played again. The X is a single letter indicating what kind of read error occurred:

'a'	Bit too long	An impossibly long one or zero has been measured. This often indicates reading past the end of the record.
'b'	CRC error	Data was read from tape incorrectly.
'd'	Block too long	The data record contains more than the expected 2048 bytes of data.

```
Write error a
```

An error occurred whilst writing to the tape. There is only one possible write error. This indicates that the Cassette Manager was unable to write a bit as fast as was requested. This error will never occur unless the user has set the write speed beyond the maximum possible.

8.12 Escape Key.

The escape key on the keyboard may be used to abandon cassette operations at certain times.

When the Cassette Manager issues one of the prompt messages it calls KM READ CHAR repeatedly to empty the key buffer out. Then it calls KM WAIT KEY to wait until the user presses a key to acknowledge the prompt. If the value generated from the key the user presses is #FC, which is the value normally generated by the escape key, then the Cassette Manager will abandon the read or write and will return an error condition to the caller.

When reading from or writing to the cassette interrupts are disabled and the normal key scanning mechanism is not active. While reading or writing the record leader the Cassette Manager itself scans the keyboard to test whether key 66, the escape key, is pressed. If this key is found to be pressed then the Cassette Manager abandons the read or write and returns to the caller (with an appropriate error condition). While reading or writing the data in the record there is no way to interrupt the Cassette Manager, thus pressing ESC may not be detected for several seconds.

8.13 Low Level Cassette Driving.

To allow the user to produce a new filing system the record read and write routines, CAS READ and CAS WRITE, are in the firmware jumpblock. There is a third routine at this level, CAS CHECK, whose facilities are not used by the higher levels of the Casssette Manager. It allows the data that has been written to tape to be compared with the data in store. This could be used to perform a read after write check if so desired.

Also available in the firmware jumpblock are routines to turn the cassette motor on and off (CAS START MOTOR and CAS STOP MOTOR). It is not necessary to turn the motor on and off around a call of CAS READ, CAS WRITE or CAS CHECK as these routines automatically turn the motor on and off.

10 Interrupts.

There is only one source of interrupts in an unexpanded machine, namely a regular time interrupt. Expansion boards may generate interrupts, but suitable software must be provided to deal with the extra interrupts.

The system runs with interrupts enabled most of the time. It is inadvisable to disable interrupts for a prolonged period if this is avoidable because the time interrupts will be missed.

A number of firmware routines enable interrupts and this is remarked upon in their descriptions. In particular the Kernel routines dealing with ROMs and the restart instructions (e.g. LOW JUMP) enable interrupts.

10.1 The Time Interrupt.

The time interrupt occurs roughly once every 1/300th of a second. On machines with PAL monitors (as in the UK) or SECAM monitors (as in France) the timer is synchronised with frame flyback every sixth tick. On machines using NSTC monitors (as in the US) the timer is synchronised with frame flyback every fifth tick. The time interrupt is processed by the Kernel and presented to the rest of the system in a number of ways:

a. Fast Ticker Interrupts. Period = 1/300th of a second.

For high resolution or very short period timing (not intended for general use).

b. Sound Generation Interrupt. Period = 1/100th of a second.

This interrupt drives the sound generation firmware, but is otherwise not visible to the system.

c. Frame Flyback Interrupt. Period = 1/50th or 1/60th of a second.

For actions which must take place during frame flyback. Ink flashing is performed during a frame flyback interrupt, for example.

d. Ticker Interrupt. Period = 1/50th of a second.

This is the general purpose ticker interrupt. The keyboard is scanned at the start of each ticker interrupt.

e. System Clock.

There is a timer that counts fast ticks i.e. 1/300ths of a second. This can be used to measure elapsed time without setting up a relatively expensive fast tick event (see section 10.5). The timer is read by calling KL TIME PLEASE and may be set by calling KL TIME SET.

10.2 External Interrupts.

The Z80 is run in interrupt mode 1. Which is to say that all interrupts cause an RST 7 to be executed by the processor. The interrupt handling code in the Kernel can distinguish between the time interrupt and an external interrupt. It does this by re-enabling interrupts inside the interrupt routine. If the interrupt repeats then it is assumed to be an external interrupt, otherwise it is taken to be a time interrupt. Note that this requires that the source of external interrupts should not clear the interrupt condition until the software resets it.

Before an external interrupt is enabled its interrupt handler must be 'installed'. This is done by copying the 5 bytes at address #003B to a new location and replacing them by suitable code (probably including a jump). When the Kernel detects an external interrupt it calls address #003B in RAM to process the interrupt:

Entry:

No conditions.

Exit:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Interrupts are disabled and must remain disabled.
The lower ROM is disabled.
The upper ROM select and state are indeterminate.
The alternate register set must not be touched.

The interrupt routine must establish whether it can deal with the interrupt, and if so it must at least clear it. If the interrupt is not the responsibility of the routine then it should jump to the copy of the bytes taken from location #003B which may be competent to deal with the interrupt. This requires the code patched at location #003B to be position independent in case a second external interrupt handler is installed. The code put at #003B at EMS is position independent - it merely returns.

Note that interrupt handling code must be in RAM somewhere between #0040 and #BFFF. Interrupt handlers should be as short as possible. If an interrupt requires a lot of processing beyond that required to clear it, then the interrupt should kick an event to do the work outside the interrupt path.

10.3 Nonmaskable Interrupts.

There is no provision for handling a nonmaskable interrupt (NMI) in the firmware (despite the fact that NMI is available on the external bus connector). Various firmware routines (notably those connected with driving the Centronics port, the PPI to access the sound chip and keyboard, and the cassette) will have timing constraints violated if NMIs occur whilst they are active. It is recommended that NMI should not be used.

10.4 Interrupts and Events.

As a general rule hardware interrupts should be transformed into their software equivalents, 'events', as soon as possible. The handling of events is more flexible than the handling of hardware interrupts - for example there are no restrictions on where event routines may reside, or on interrupt enabling.

Events are described by an event block. This block contains the event class, the event count and an event routine address. When an event occurs the event block is 'kicked' and the Kernel arranges for the event routine to be called once for each kick (the number of kicks outstanding is kept in the event block). The event routine is not necessarily called immediately. When the event routine is actually run depends on the event class as follows:

a. Express Asynchronous Events.

This is an unusual class of event. The event routine is called immediately during interrupt processing. The routine must be accessible by the interrupt code, it may not enable interrupts, corrupt the IX or IY registers or use the alternate register set. The routine should be as short as possible.

b. Normal Asynchronous Events.

This is the most flexible sort of event. When the event is kicked the event routine is not called, but the event block is placed on the interrupt event pending queue.

Once the current interrupt has been processed, just before the Kernel returns from the interrupt path, any events on the interrupt event pending queue are processed. While the events are being processed the system is running with interrupts enabled and may be regarded as no longer being in the interrupt path. It is using its own stack rather than the main system stack. This private stack is 128 bytes long.

The asynchronous event routine is, therefore, called shortly after the event is kicked and is not restricted in what it may do or where it may be located. The event routine may take as long to run as is needed. Any further kicks received during the time that the event routine is running will be added to the event count and will be processed before returning to the interrupted program.

c. Synchronous Events.

Synchronous events are queued on the synchronous event pending queue. They are not processed until the foreground program allows the queue to be processed. This can be used to control interactions between different parts of programs.

10.5 Interrupt Queues.

The various time interrupts provide three sources of 'kicks' for events. The events to be kicked when each of the interrupts occur are stored on queues, one queue for each source of kicks. The user provides an area of store for the Kernel's use. The size of the area depends on which queue it is for. The last 7 bytes of the area are always an event block which the user should initialise appropriately.

a. Fast Ticker Events.

Events on the fast ticker queue are 'kicked' on each fast ticker interrupt, i.e. every 1/300th of a second. A fast ticker block is 9 bytes long.

b. Ticker Events.

Each event on the ticker queue is associated with a timer. The timer may be a 'one shot', which goes off once, or a repeater, which goes off periodically. The timer counts ticker interrupts, i.e. 1/50ths of a second, and when sufficient have occurred it goes off. Each time the timer associated with an event goes off the event is kicked. A ticker block is 13 bytes long.

c. Frame Flyback Events.

Events on the frame flyback queue are kicked on each frame flyback interrupt, i.e. every 1/50th of a second on PAL or SECAM machines and every 1/60th of a second on NSTC machines. A frame flyback block is 9 bytes long.

11 Events.

The event mechanism is primarily provided by the Kernel to support the handling of interrupts and other external events. However, the mechanism may also be used to handle internal events in complicated programs (such as a simulation, for example). An event is characterised by the following:

a. **Event Class** (see section 11.1)

Events may be synchronous or asynchronous, express or normal.

b. **Event Priority** (see section 11.1)

Synchronous events have an associated priority.

c. **Event Count** (see section 11.2)

Each time an event occurs the count is incremented.
Each time an event is processed the count is decremented.
The event may be disarmed by setting the count negative.

d. **Event Routine** (see section 11.3)

The address of the routine which is called to process the event.

An event appears to the Kernel as a data block containing the above values (see Appendix X for the exact layout of an event block). The block must be in the central 32K bytes of memory, so that the Kernel can access it without worrying about the ROM enable state.

When an event occurs the associated event block is kicked by calling KL EVENT. If the event count is negative, the 'kick' is ignored, otherwise the event count is incremented (up to a maximum of 127) and the event routine will be called at some time in the future - depending on the event class. When the event routine returns the event count is decremented, unless it has been set to zero or negative in the meantime.

11.1 Event Class.

Events are either synchronous or asynchronous. Asynchronous events are intended for the processing of external events which require almost immediate service. The processing of asynchronous events pre-empts the main program. The processing of synchronous events is under the complete control of the main program, which will, in general, deal with them when it is convenient to do so.

a. Asynchronous Events.

An asynchronous event is processed immediately the event is kicked - or almost immediately if the kick occurs in the interrupt path - see section 10 on interrupts. The Kernel does not provide any interlocks between asynchronous events and the main program or other events, so care must be exercised to avoid interactions. It is most unwise to call routines that are not re-entrant - for example, the firmware screen driving routines.

If the event count is still greater than zero when the event routine returns, it is decremented. If the count remains greater than zero then the process is repeated (the event routine is called again and the event count is decremented) until the count becomes zero or is set negative (see 11.2 below).

b. Synchronous Events.

Synchronous events are not processed when the event is kicked, but are placed on the synchronous event queue, waiting to be processed. Events are queued in descending order of priority - equal priority events after those already on the queue.

The foreground program should poll the synchronous event queue regularly, to see if there are any events outstanding. If there are then it should then process them. The difference between synchronous and asynchronous events is, therefore, that the foreground program decides when synchronous events should be processed, but the event 'kicker' decides when asynchronous events are to be processed. Provided that the foreground program takes suitable care, there should be no difficulty in handling the interactions and resource sharing between synchronous events and the foreground program.

When the foreground program finds the synchronous event queue is not empty it should (but is not constrained to) instruct the Kernel to process the first event on the queue. When a synchronous event routine is run the Kernel remembers the priority of the event. In the event routine the synchronous event queue may be polled, but the Kernel hides any events whose priority is less than or equal to that of the event currently being processed. When the event routine returns the previous event priority is restored - so the processing of events may be nested.

The synchronous event priorities are split into two ranges, express and normal. All express events have higher priorities than all normal events. The Kernel provides a mechanism to disable the processing of normal events, without affecting express events. This may be used to implement 'critical regions' through which normal events may interact. The synchronous event 'kicked' by the Key Manager break handling mechanism is an example of an express synchronous event.

11.2 Event Count.

The main purpose of the event count is to keep track of the difference between the number of times the event has been kicked, and the number of times the event has been processed. This ensures that a kick is not missed if it occurs before the previous kick has been processed. The event count is normally incremented when the event is kicked and decremented when the event routine returns. However the exact action depends on the event count as follows:

Increment.

−128..−2:	The count is not changed - the event is ignored.
−1:	This value is illegal.
0:	The count is incremented and event processing is initiated as required by the event class.
1..126:	The count is incremented but no further action is taken. The event is waiting for a previous kick to be processed or for processing to complete.
127:	The count is not changed - the kick is ignored.

Decrement.

−128:	This value is illegal.
−127..0:	The count is not changed - the event has been disarmed.
1:	The count is decremented and event processing is terminated.
2..127:	The count is decremented and event processing continues.

Note that the event routine may disarm itself by setting the count negative (by convention to −64) and can discard unwanted kicks by setting its count to one.

11.3 Event Routine.

In general the address of the event routine is given as a 3 byte 'far address' (see section 2 on the memory layout). This allows the routine to be located in any ROM or anywhere in RAM.

A special form of the event class may specify the routine as at a 'near address'. This does not change the ROM state and so the routine must be located either in the lower ROM or in the central 32K of RAM. The ROM select byte of the 'far address' is ignored and the other two bytes taken as the address of the routine. Calling a 'near address' event routine requires a little less work than calling a full 'far address', and is used by the firmware itself.

11.4 Disarming and Reinitialising Events.

Before an event block may be reinitialised the event must be disarmed. This ensures that the event is removed from the various event pending queues and prevents the event queues being corrupted when the event block is initialised. An asynchronous event must not be reinitialised from inside its asynchronous event routine (because in this case disarming the event does not remove the event from the interrupt event pending queue).

Synchronous and asynchronous events are disarmed in different manners.

a. Asynchronous Events.

An asynchronous event should be disarmed by calling KL DISARM EVENT. This sets the event count to a negative value (-64) and thus prevents kicks having any effect. If the event is on the interrupt event pending queue then it will be discarded only when an attempt is made to process the event and not immediately that the event is disarmed.

b. Synchronous Events.

A synchronous event should be disarmed by calling KL DEL SYNCHRONOUS. This sets the event count to a negative value (-64) and removes the event block from the synchronous event pending queue (if it is on the queue).

The above procedures prevent the event being successfully kicked, they do not prevent attempts being made to kick the event. A fast ticker, frame flyback or ticker event (see section 4.5) will still be on its appropriate queue and will still be receiving regular attempts to kick it. To prevent time being wasted (and the system from being slowed down because of it) the event should be removed from the interrupt queue by calling KL DEL FAST TICKER, KL DEL FRAME FLY or KL DEL TICKER.

12 The Machine Pack.

The Machine Pack deals with the low level driving of the hardware. It also talks to the Centronics port (and hence the printer) and is in charge of running 'load and go' programs.

12.1 Hardware Interfaces.

The routines provided for driving the hardware are only to be used by those who understand the hardware and how the firmware drives the hardware. The user should not access the hardware directly when a Machine Pack routine is provided for this purpose.

Often there are higher level routines that accomplish the same effects but that also keep the firmware informed of the current settings. Where possible these higher level routines should be used and the Machine Pack routines avoided. Using the Machine Pack routines may cause the firmware to make erroneous assumptions about the current settings and may cause it to go wrong.

The Machine Pack makes certain assumptions about the state of the hardware when it accesses it. In particular, PPI port A is assumed to be in output mode and the sound chip, ULA, CRTC and Centronics port are assumed to be inactive; that is, not half way through setting a value into a chip register. It is usually essential that interrupts be disabled when accessing the hardware directly.

There are four main areas of the hardware that the Machine Pack deals with:

a. The screen.

There are three aspects of the screen display that can be set using Machine Pack routines. These are the screen mode (set by calling MC SET MODE) and the screen base and offset (set by calling MC SET OFFSET).

The screen mode sets how many pixels are displayed on the screen and how many inks may be used as follows:

Mode	Resolution	Inks
0	160 x 200	16
1	320 x 200	4
2	640 x 200	2

The screen base sets which 16K block of memory is used for the screen memory. Theoretically, any of #0000, #4000, #8000 or #C000 could be used but, in practice, other considerations mean that only #4000 and #C000 are useful.

The screen offset sets which byte in the screen memory is to be displayed first. Changing the offset will move the contents of the screen in one go. This is used for rolling the screen.

A fuller description of the screen layout and its relationship to these aspects can be found in section 7 on the Screen Pack.

If addresses are to be read back from the CRT controller chip, when using a light pen for instance, then careful inspection of the way the screen memory is addressed will be needed to translate the screen address read from the chip to the actual position on the screen.

The Machine Pack also provides a routine (MC WAIT FLYBACK) to wait until frame flyback occurs (the start of the vertical retrace period). This may be used to ensure that operations on the screen are performed with as little disruption as is possible to the picture on the monitor since no picture is generated during this period. As an alternative to waiting for frame flyback explicitly the user should consider setting up a frame flyback event as described in section 10.5.

The vertical retrace period is not very long. Furthermore, approximately 100 microseconds from its start, a time interrupt occurs that will cause the frame flyback events to be processed (see section 10). These may take a significant length of time out of the retrace period.

b. The inks.

The Machine Pack deals with setting the colours of inks. There is a fuller explanation of the relationship between inks and colours in section 6.2. Briefly, the colour for each ink and the border can be specified independently and changed at will. Note, however, that the Machine Pack deals with the hardware representations of colours and not the grey scale colours that the Screen Pack uses and also that an ink may only be set to one colour, the flashing inks are made by the Screen Pack setting two colours alternately.

Two routines are provided for setting the colours of inks. MC SET INKS allows the colours of all 16 inks and the border to be set (although not all of the inks may be visible on the screen in the current mode). MC CLEAR INKS sets the colour of the border ink and sets all 16 inks to the same colour. The latter is used when clearing the screen to make the operation appear instantaneous.

c. The sound chip.

A routine, MC SOUND REGISTER, is provided to write to a register of the sound chip. This is used by the Sound Manager for hardware access.

d. The Centronics port.

Two routines are provided to access the Centronics port. MC BUSY PRINTER tests if it is busy. MC SEND PRINTER strobes data out of it. Data should not be sent while the port is busy.

The Centronics port is used by the printer routines provided in the Machine Pack and described below.

12.2 The Printer.

There is a routine, MC PRINT CHAR, which calls an indirection, MC WAIT PRINTER, for sending characters to the printer, or rather, to the Centronics port.

MC WAIT PRINTER waits until the Centronics port is not busy and then sends the given character to it. If the port remains busy for a long time then the routine times out and returns indicating that it has failed to send the character. This time out can be used to prevent programs 'hanging' because they are waiting for a (possibly non-existent) printer to become ready.

MC WAIT PRINTER allows the user to intercept characters to be sent to the printer. This could allow special escape sequences to be inserted if needed, or it could allow the printer to be disabled or the length of the time out to be changed.

12.3 Loading and Running Programs.

The Machine Pack provides two routines for running programs, MC START PROGRAM and MC BOOT PROGRAM.

MC START PROGRAM is the simpler of the two routines. It completely re-initialises all the firmware and then enters the given program.

MC BOOT PROGRAM is more complex. It is for loading a program into RAM and running it. The user supplies a routine to MC BOOT PROGRAM that will load the program and return its entry point. Before this load routine is called as much of the firmware as is possible is reset so that the area of memory between #0040 and the base of the firmware RAM at #B100 is available for use. If the system were not reset then an active indirection, event or interrupt routine might be overwritten with disastrous consequences.

If the program is loaded successfully by MC BOOT PROGRAM then the firmware is completely initialised and the program is entered. However, if the loading fails then an appropriate message is printed and the previous foreground program is restarted. If the previous program was itself a RAM program then the default ROM is entered instead because it is likely that the previous program was corrupted when the attempt to load the new one was made.

13 Firmware Jumpblocks.

There are a number of jumpblocks provided by the firmware. The largest of these is the main firmware jumpblock. This is intended to be used by programs to access the firmware routines in the lower ROM. BASIC, for instance, uses these jumps. Note, however that the firmware does not use this jumpblock for internal communication with itself. This means that altering the jumpblock will cause BASIC to behave differently but will not cause the firmware to behave differently.

The next most important jumpblock is the indirections jumpblock. The indirections are jumps that are used by the firmware at key points. This allows the user to alter the action of firmware routines. The entries in this jumpblock are not intended for the user to call, only for the firmware to call. Altering an indirection is the method to make the firmware behave differently.

The remaining two jumpblocks are associated with the Kernel. One is a jumpblock to allow the user to call various useful Kernel routines to do with changing ROM states and the like. The other is not a jumpblock as such, just an area where the routines are at published addresses. These are general utility routines and restarts. In general neither of these areas should be altered by the user.

The routines in these jumpblocks are briefly listed below. More complete descriptions of the routines can be found in sections 14, 15 and 16.

13.1 The Main Jumpblock.

The main firmware jumpblock lies in RAM between addresses #BB00 and #BD39. Each entry in the jumpblock occupies three bytes and is initialised to use LOW JUMP restarts (RST1) that cause the lower ROM to be enabled, so that the firmware routines can be run, and the upper ROM to be disabled, so that the screen memory is accessible while the firmware is running.

After the jumpblock has been set up at EMS it is not altered by the firmware until the system is reinitialised. If any entries are changed then it is the user's responsibility to undo the alterations. This can be achieved by calling JUMP RESTORE which completely initialises the jumpblock.

13.1.1 Entries to the Key Manager

The Key Manager deals with the keyboard and the joysticks.

INITIALISATION

0	#BB00	KM INITIALISE	Initialise the Key Manager.

1	#BB03	KM RESET	Reset the Key Manager - clear all buffers, restore standard key expansions and indirections.

CHARACTERS

2	#BB06	KM WAIT CHAR	Wait for next character from the keyboard.
3	#BB09	KM READ CHAR	Test if a character is available from the keyboard.
4	#BB0C	KM CHAR RETURN	Return a single character to the keyboard for next time.
5	#BB0F	KM SET EXPAND	Set an expansion string.
6	#BB12	KM GET EXPAND	Get a character from an expansion string.
7	#BB15	KM EXP BUFFER	Allocate a buffer for expansion strings.

KEYS

8	#BB18	KM WAIT KEY	Wait for next key from the keyboard.
9	#BB1B	KM READ KEY	Test if a key is available from the keyboard.
10	#BB1E	KM TEST KEY	Test if a key is pressed.
11	#BB21	KM GET STATE	Fetch Caps Lock and Shift Lock states.
12	#BB24	KM GET JOYSTICK	Fetch current state of the joystick(s).

TRANSLATION TABLES

13	#BB27	KM SET TRANSLATE	Set entry in key translation table without shift or control.
14	#BB2A	KM GET TRANSLATE	Get entry from key translation table without shift or control.
15	#BB2D	KM SET SHIFT	Set entry in key translation table when shift key is pressed.
16	#BB30	KM GET SHIFT	Get entry from key translation table when shift key is pressed.
17	#BB33	KM SET CONTROL	Set entry in key translation table when control key is pressed.

18	#BB36	KM GET CONTROL	Get entry from key translation table when control key is pressed.

REPEATING

19	#BB39	KM SET REPEAT	Set whether a key may repeat.
20	#BB3C	KM GET REPEAT	Ask if a key is allowed to repeat.
21	#BB3F	KM SET DELAY	Set start up delay and repeat speed.
22	#BB42	KM GET DELAY	Get start up delay and repeat speed.

BREAKS

23	#BB45	KM ARM BREAK	Allow break events to be generated.
24	#BB48	KM DISARM BREAK	Prevent break events from being generated.
25	#BB4B	KM BREAK EVENT	Generate a break event (if armed).

13.1.2 Entries to the Text VDU

The Text VDU is a character based screen driver.

INITIALISATION

26	#BB4E	TXT INITIALISE	Initialise the Text VDU.
27	#BB51	TXT RESET	Reset the Text VDU - restore default indirections and control code functions.
28	#BB54	TXT VDU ENABLE	Allow characters to be placed on the screen.
29	#BB57	TXT VDU DISABLE	Prevent characters from being placed on the screen.

CHARACTERS

30	#BB5A	TXT OUTPUT	Output a character or control code to the Text VDU.
31	#BB5D	TXT WR CHAR	Write a character onto the screen.
32	#BB60	TXT RD CHAR	Read a character from the screen.

33	#BB63	TXT SET GRAPHIC	Turn on or off the Graphics VDU character writing option.

WINDOWS

34	#BB66	TXT WIN ENABLE	Set the size of the current text window.
35	#BB69	TXT GET WINDOW	Get the size of the current text window.
36	#BB6C	TXT CLEAR WINDOW	Clear current window.

CURSOR

37	#BB6F	TXT SET COLUMN	Set cursor horizontal position.
38	#BB72	TXT SET ROW	Set cursor vertical position.
39	#BB75	TXT SET CURSOR	Set cursor position.
40	#BB78	TXT GET CURSOR	Ask current cursor position.
41	#BB7B	TXT CUR ENABLE	Allow cursor display - user.
42	#BB7E	TXT CUR DISABLE	Disallow cursor display - user.
43	#BB81	TXT CUR ON	Allow cursor display - system.
44	#BB84	TXT CUR OFF	Disallow cursor display -system.
45	#BB87	TXT VALIDATE	Check if a cursor position is within the window.
46	#BB8A	TXT PLACE CURSOR	Put a cursor blob on the screen.
47	#BB8D	TXT REMOVE CURSOR	Take a cursor blob off the screen.

INKS

48	#BB90	TXT SET PEN	Set ink for writing characters.
49	#BB93	TXT GET PEN	Get ink for writing characters.
50	#BB96	TXT SET PAPER	Set ink for writing text background.
51	#BB99	TXT GET PAPER	Get ink for writing text background.
52	#BB9C	TXT INVERSE	Swap current pen and paper inks.
53	#BB9F	TXT SET BACK	Allow or disallow background being written.

54	#BBA2	TXT GET BACK	Ask if background is being written.

MATRICES

55	#BBA5	TXT GET MATRIX	Get the address of a character matrix.
56	#BBA8	TXT SET MATRIX	Set a character matrix.
57	#BBAB	TXT SET M TABLE	Set the user defined matrix table address.
58	#BBAE	TXT GET M TABLE	Get user defined matrix table address.

CONTROL CODES

59	#BBB1	TXT GET CONTROLS	Fetch address of control code table.

STREAMS

60	#BBB4	TXT STR SELECT	Select a Text VDU stream.
61	#BBB7	TXT SWAP STREAMS	Swap the states of two streams.

13.1.3 Entries to the Graphics VDU

The Graphics VDU deals with individual pixels.

INITIALISATION

62	#BBBA	GRA INITIALISE	Initialise the Graphics VDU.
63	#BBBD	GRA RESET	Reset the Graphics VDU -restore standard indirections.

CURRENT POSITION

64	#BBC0	GRA MOVE ABSOLUTE	Move to an absolute position.
65	#BBC3	GRA MOVE RELATIVE	Move relative to current position.
66	#BBC6	GRA ASK CURSOR	Get the current position.
67	#BBC9	GRA SET ORIGIN	Set the origin of the user coordinates.
68	#BBCC	GRA GET ORIGIN	Get the origin of the user coordinates.

WINDOW

69	#BBCF	GRA WIN WIDTH	Set left and right edges of the graphics window.
70	#BBD2	GRA WIN HEIGHT	Set the top and bottom edges of the graphics window.
71	#BBD5	GRA GET W WIDTH	Get the left and right edges of the graphics window.
72	#BBD8	GRA GET W HEIGHT	Get the top and bottom edges of the graphics window.
73	#BBDB	GRA CLEAR WINDOW	Clear the graphics window.

INKS

74	#BBDE	GRA SET PEN	Set the graphics plotting ink.
75	#BBE1	GRA GET PEN	Get the current graphics plotting ink.
76	#BBE4	GRA SET PAPER	Set the graphics background ink.
77	#BBE7	GRA GET PAPER	Get the current graphics background ink.

PLOTTING

78	#BBEA	GRA PLOT ABSOLUTE	Plot a point at an absolute position.
79	#BBED	GRA PLOT RELATIVE	Plot a point relative to the current position.

TESTING

80	#BBF0	GRA TEST ABSOLUTE	Test a point at an absolute position.
81	#BBF3	GRA TEST RELATIVE	Test a point relative to the current position.

LINE DRAWING

82	#BBF6	GRA LINE ABSOLUTE	Draw a line to an absolute position.
83	#BBF9	GRA LINE RELATIVE	Draw a line relative to the current position.

CHARACTER DRAWING

84	#BBFC	GRA WR CHAR	Put a character on the screen at the current graphics position.

13.1.4 Entries to the Screen Pack

The Screen Pack interfaces the Text and Graphic VDUs to the screen hardware. Screen functions that affect both text and graphics (e.g. ink colours) are located in the Screen Pack.

INITIALISATION

85	#BBFF	SCR INITIALISE	Initialise the Screen Pack.
86	#BC02	SCR RESET	Reset the Screen Pack - restore standard indirections, ink colours and flash rates.

SCREEN HARDWARE

87	#BC05	SCR SET OFFSET	Set the offset of the start of the screen.
88	#BC08	SCR SET BASE	Set the area of RAM to use for the screen memory.
89	#BC0B	SCR GET LOCATION	Fetch current base and offset settings.

MODE

90	#BC0E	SCR SET MODE	Set screen into a new mode.
91	#BC11	SCR GET MODE	Ask the current screen mode.
92	#BC14	SCR CLEAR	Clear the screen (to ink zero).
93	#BC17	SCR CHAR LIMITS	Ask size of the screen in characters.

SCREEN ADDRESSES

94	#BC1A	SCR CHAR POSITION	Convert physical coordinates to a screen position.
95	#BC1D	SCR DOT POSITION	Convert base coordinates to a screen position.
96	#BC20	SCR NEXT BYTE	Step a screen address right one byte.

97	#BC23	SCR PREV BYTE	Step a screen address left one byte.
98	#BC26	SCR NEXT LINE	Step a screen address down one line.
99	#BC29	SCR PREV LINE	Step a screen address up one line.

INKS

100	#BC2C	SCR INK ENCODE	Encode an ink to cover all pixels in a byte.
101	#BC2F	SCR INK DECODE	Decode an encoded ink.
102	#BC32	SCR SET INK	Set the colours in which to display an ink.
103	#BC35	SCR GET INK	Ask the colours an ink is currently displayed in.
104	#BC38	SCR SET BORDER	Set the colours in which to display the border.
105	#BC3B	SCR GET BORDER	Ask the colours the border is currently displayed in.
106	#BC3E	SCR SET FLASHING	Set the flash periods.
107	#BC41	SCR GET FLASHING	Ask the current flash periods.

MISCELLANEOUS

108	#BC44	SCR FILL BOX	Fill a character area of the screen with an ink.
109	#BC47	SCR FLOOD BOX	Fill a byte area of the screen with an ink.
110	#BC4A	SCR CHAR INVERT	Invert a character position.
111	#BC4D	SCR HW ROLL	Move the whole screen up or down eight pixel lines (one character).
112	#BC50	SCR SW ROLL	Move an area of the screen up or down eight pixel lines (one character).
113	#BC53	SCR UNPACK	Expand a character matrix for the current screen mode.
114	#BC56	SCR REPACK	Compress a character matrix to the standard form.

115	#BC59	SCR ACCESS	Set the screen write mode for the Graphics VDU.
116	#BC5C	SCR PIXELS	Write a pixel to the screen ignoring the Graphics VDU write mode.

LINE DRAWING

117	#BC5F	SCR HORIZONTAL	Plot a purely horizontal line
118	#BC62	SCR VERTICAL	Plot a purely vertical line.

13.1.5 Entries to the Cassette Manager

The Cassette Manager handles reading files from tape and writing files to tape.

INITIALISATION

119	#BC65	CAS INITIALISE	Initialise the Cassette Manager - close all streams, set default speed and enable messages.
120	#BC68	CAS SET SPEED	Set the write speed.
121	#BC6B	CAS NOISY	Enable or disable prompt messages.

MOTOR CONTROL

122	#BC6E	CAS START MOTOR	Start the cassette motor.
123	#BC71	CAS STOP MOTOR	Stop the cassette motor.
124	#BC74	CAS RESTORE MOTOR	Restore previous state of cassette motor.

READING FILES

125	#BC77	CAS IN OPEN	Open a file for input.
126	#BC7A	CAS IN CLOSE	Close the input file properly.
127	#BC7D	CAS IN ABANDON	Close the input file immediately.
128	#BC80	CAS IN CHAR	Read a character from the input file.
129	#BC83	CAS IN DIRECT	Read the input file into store.

130	#BC86	CAS RETURN	Put the last character read back.
131	#BC89	CAS TEST EOF	Have we reached the end of the input file yet?

WRITING FILES

132	#BC8C	CAS OUT OPEN	Open a file for output.
133	#BC8F	CAS OUT CLOSE	Close the output file properly.
134	#BC92	CAS OUT ABANDON	Close the output file immediately.
135	#BC95	CAS OUT CHAR	Write a character to the output file.
136	#BC98	CAS OUT DIRECT	Write the output file directly from store.

CATALOGUING

137	#BC9B	CAS CATALOG	Generate a catalogue from the tape.

RECORDS

138	#BC9E	CAS WRITE	Write a record to tape.
139	#BCA1	CAS READ	Read a record from tape.
140	#BCA4	CAS CHECK	Compare a record on tape with the contents of store.

13.1.6 Entries to the Sound Manager

The Sound Manager controls the sound chip.

INITIALISATION

141	#BCA7	SOUND RESET	Reset the Sound Manager -shut the sound chip up and clear all sound queues.

SOUND QUEUES

142	#BCAA	SOUND QUEUE	Add a sound to a sound queue.
143	#BCAD	SOUND CHECK	Ask if there is space in a sound queue.

144	#BCB0	SOUND ARM EVENT	Set up an event to be run when a sound queue becomes not full.

SOUNDS

145	#BCB3	SOUND RELEASE	Allow sounds to happen.
146	#BCB6	SOUND HOLD	Stop all sounds in mid flight.
147	#BCB9	SOUND CONTINUE	Restart sounds after they have been stopped.

ENVELOPES

148	#BCBC	SOUND AMPL ENVELOPE	Set up an amplitude envelope.
149	#BCBF	SOUND TONE ENVELOPE	Set up a tone envelope.
150	#BCC2	SOUND A ADDRESS	Get the address of an amplitude envelope.
151	#BCC5	SOUND T ADDRESS	Get the address of a tone envelope.

13.1.7 Entries to the Kernel

The Kernel handles synchronous and asynchronous events. It is also in charge of the store map and switching ROMs on and off. Apart from the entries listed below, the Kernel has its own jumpblock and a number of routines whose addresses are published. These extra entries are listed in sections 13.3 and 13.4 below.

INITIALISATION

152	#BCC8	KL CHOKE OFF	Reset the Kernel - clears all event queues etc.
153	#BCCB	KL ROM WALK	Find and initialise all background ROMs.
154	#BCCE	KL INIT BACK	Initialise a particular background ROM.
155	#BCD1	KL LOG EXT	Introduce an RSX to the firmware.
156	#BCD4	KL FIND COMMAND	Search for an RSX or background ROM or foreground ROM to process a command.

FRAME FLYBACK LIST

157	#BCD7	KL NEW FRAME FLY	Initialise and put a block onto the frame flyback list.
158	#BCDA	KL ADD FRAME FLY	Put a block onto the frame flyback list.
159	#BCDD	KL DEL FRAME FLY	Remove a block from the frame flyback list.

FAST TICK LIST

160	#BCE0	KL NEW FAST TICKER	Initialise and put a block onto the fast tick list.
161	#BCE3	KL ADD FAST TICKER	Put a block onto the fast tick list.
162	#BCE6	KL DEL FAST TICKER	Remove a block from the fast tick list.

TICK LIST

163	#BCE9	KL ADD TICKER	Put a block onto the tick list.
164	#BCEC	KL DEL TICKER	Remove a block from the tick list.

EVENTS

165	#BCEF	KL INIT EVENT	Initialise an event block.
166	#BCF2	KL EVENT	'Kick' an event block.
167	#BCF5	KL SYNC RESET	Clear synchronous event queue.
168	#BCF8	KL DEL SYNCHRONOUS	Remove a synchronous event from the event queue.
169	#BCFB	KL NEXT SYNC	Get the next event from the queue.
170	#BCFE	KL DO SYNC	Perform an event routine.
171	#BD01	KL DONE SYNC	Finish processing an event.
172	#BD04	KL EVENT DISABLE	Disable normal synchronous events.
173	#BD07	KL EVENT ENABLE	Enable normal synchronous events.
174	#BD0A	KL DISARM EVENT	Prevent an event from occurring.

ELAPSED TIME

175	#BD0D	KL TIME PLEASE	Ask the elapsed time.
176	#BD10	KL TIME SET	Set the elapsed time.

13.1.8 Entries to the Machine Pack

The Machine Pack provides an interface to the machine hardware. Most packs use Machine to access any hardware they use. The major exception is the Cassette Manager which, for speed reasons, performs its own hardware access.

PROGRAMS

177	#BD13	MC BOOT PROGRAM	Load and run a foreground program.
178	#BD16	MC START PROGRAM	Run a foreground program.

SCREEN

179	#BD19	MC WAIT FLYBACK	Wait for frame flyback.
180	#BD1C	MC SET MODE	Set the screen mode.
181	#BD1F	MC SCREEN OFFSET	Set the screen offset.
182	#BD22	MC CLEAR INKS	Set all inks to one colour.
183	#BD25	MC SET INKS	Set colours of all the inks.

PRINTER

184	#BD28	MC RESET PRINTER	Reset the printer indirection.
185	#BD2B	MC PRINT CHAR	Try to send a character to the Centronics port.
186	#BD2E	MC BUSY PRINTER	Test if the Centronics port is busy.
187	#BD31	MC SEND PRINTER	Send a character to the Centronics port.

SOUND CHIP

188	#BD34	MC SOUND REGISTER	Send data to a sound chip register.

13.1.9 Entries to Jumper

Jumper sets up the main jumpblock.

INITIALISATION

189	#BD37	JUMP RESTORE	Restore the standard jumpblock.

13.2 Firmware Indirections.

The firmware indirections listed here are taken at key points in the firmware thus allowing the user to provide substitute routines for many firmware actions, without having to replace a complete firmware package. These indirections are not intended for the user to call - there is usually a higher level routine in the main firmware jumpblock that is more suitable.

The indirections are set up by the pack to whom they apply whenever its reset (or initialise) routine is called and during EMS; they are not otherwise altered by the firmware.

The indirections are all three bytes long and use standard jump instructions (#C3). If a ROM state other than upper ROMs disabled and lower ROM enabled is required then the appropriate restart instruction might be substituted (see section 2.3). The indirections are to be found between #BDCD and #BDF3.

At this level of operation very little validation is carried out. If incorrect parameters are passed or a substitute routine corrupts a register in defiance of the documented interface then the firmware will probably cease to function as expected.

More detailed descriptions of these routines can be found in section 15.

13.2.1 Text VDU Indirections

0	#BDCD	TXT DRAW CURSOR	Place the cursor blob on the screen (if enabled).
1	#BDD0	TXT UNDRAW CURSOR	Remove the cursor blob from the screen (if enabled).
2	#BDD3	TXT WRITE CHAR	Write a character onto the screen.
3	#BDD6	TXT UNWRITE	Read a character from the screen.
4	#BDD9	TXT OUT ACTION	Output a character or control code.

13.2.2 Graphics VDU Indirections

5	#BDDC	GRA PLOT	Plot a point.
6	#BDDF	GRA TEST	Test a point.
7	#BDE2	GRA LINE	Draw a line.

13.2.3 Screen Pack Indirections

8	#BDE5	SCR READ	Read a pixel from the screen.
9	#BDE8	SCR WRITE	Write pixel(s) to the screen using the current graphics write mode.
10	#BDEB	SCR MODE CLEAR	Clear the screen to ink 0.

13.2.4 Keyboard Manager Indirections

11	#BDEE	KM TEST BREAK	Test for break (or reset).

13.2.5 Machine Pack Indirections

12	#BDF1	MC WAIT PRINTER	Print a character or time out.

13.3 The High Kernel Jumpblock.

The high Kernel jumpblock is provided to allow the user to turn ROMs on and off and to access memory underneath ROMs while they are enabled. The entries in this jumpblock are not all jump instructions, some entries are the start of routines, thus the user should not alter any of the entries in this jumpblock. The high Kernel jumpblock occupies store from #B900 upwards. More detailed descriptions of the routines in it can be found in section 16.

0	#B900	KL U ROM ENABLE	Turn on the current upper ROM.
1	#B903	KL U ROM DISABLE	Turn off the upper ROM.
2	#B906	KL L ROM ENABLE	Turn on the lower ROM.
3	#B909	KL L ROM DISABLE	Turn off the lower ROM.
4	#B90C	KL ROM RESTORE	Restore the previous ROM state.
5	#B90F	KL ROM SELECT	Select a particular upper ROM.

6	#B912	KL CURR SELECTION	Ask which upper ROM is currently selected.
7	#B915	KL PROBE ROM	Ask class and version of a ROM.
8	#B918	KL ROM DESELECT	Restore the previous upper ROM selection.
9	#B91B	KL LDIR	Move store (LDIR) with ROMs disabled.
10	#B91E	KL LDDR	Move store (LDDR) with ROMs disabled.
11	#B921	KL POLL SYNCHRONOUS	Check if an event with higher priority than the current event is pending.

13.4 The Low Kernel Jumpblock.

The Kernel provides a number of useful routines in the area of memory between #0000 and #003F. These are available, in some cases, both as a published routine address and as a restart instruction. In general the routines are available both in ROM and in RAM so whether the lower ROM is enabled does not matter. There are also a couple of areas available for the user to patch to trap RST 6s and interrupts from external hardware.

The low Kernel jumpblock is not intended for the user to alter. However, it may be necessary to alter it under certain circumstances. In particular a program may need to intercept the INTERRUPT ENTRY (by patching the jump at #0038) or the RESET ENTRY (by patching the bytes from #0000..#0007). If a program does change any locations in this jumpblock (other than those in the USER RESTART or EXT INTERRUPT areas) then it is the program's responsibility to ensure that the lower ROM is enabled or the original contents are restored when any other program runs. In particular the program must sort out the state when interrupts occur (hence the need to patch the INTERRUPT ENTRY).

More detailed descriptions of the routines in this jumpblock can be found in section 17.

#0000	RST 0	RESET ENTRY	Completely reset the machine as if powered up.
#0008	RST 1	LOW JUMP	Jump to lower ROM or RAM, takes an inline 'low address' to jump to.
#000B		KL LOW PCHL	Jump to lower ROM or RAM, HL contains the 'low address' to jump to.
#000E		PCBC INSTRUCTION	Jump to address in BC.

#0010	RST 2	SIDE CALL	Call to a sideways ROM, takes inline 'side address' to call.
#0013		KL SIDE PCHL	Call to a sideways ROM, HL contains 'side address' to call.
#0016		PCDE INSTRUCTION	Jump to address in DE
#0018	RST 3	FAR CALL	Call a routine in any ROM or RAM, takes an inline address of the 'far address' to call.
#001B		KL FAR PCHL	Call a routine in any ROM or RAM, C and HL contain the 'far address' to call.
#001E		PCHL INSTRUCTION	Jump to address in HL.
#0020	RST 4	RAM LAM	LD A,(HL) with all ROMs disabled.
#0023		KL FAR ICALL	Call a routine in any ROM or RAM, HL points at the 'far address' to call.
#0028	RST 5	FIRM JUMP	Jump to lower ROM, takes an inline address to jump to.
#0030	RST 6	USER RESTART	ROM version saves current ROM state in #002B, turns the lower ROM off and jumps to the RAM version. RAM version may be patched by the user between #0030 and #0037 inclusively.
#0038	RST 7	INTERRUPT ENTRY	This restart is not available as it is used for interrupts (Z80 interrupt mode 1).
#003B		EXT INTERRUPT	When an interrupt occurs on the expansion port the firmware calls location #003B in RAM. The user may patch between #003B and #003F inclusive to trap this occurence.

BB00
↓
BC8C

Routines du Firmware

14 The Main Firmware Jumpblock.

This section describes in detail the entry and exit conditions and the effects of all the routines in the main firmware jumpblock. The main firmware jumpblock is described in section 13.1.

The user is advised to read the sections on each pack before attempting to understand the jumpblock entries. The relevant sections are:

Key Manager	(KM)	Section 3.
Text VDU	(TXT)	Section 4.
Graphics VDU	(GRA)	Section 5.
Screen Pack	(SCR)	Section 6.
Sound Manager	(SOUND)	Section 7.
Cassette Manager	(CAS)	Section 8.
Kernel	(KL)	Sections 2, 9, 10 and 11.
Machine Pack	(MC)	Section 12.

The top line of each description has the following layout:

<Entry number>: <Entry name> <Entry address>

Entries in the jumpblock are numbered starting from zero. The entry address is the address to call to invoke the firmware routine or the address of the three bytes to patch to intercept the routine. The entry address can be calculated as:

Entry address = Start of jumpblock + 3 * Entry number

Each entry is named and is refered to by name throughout this manual.

The last section of each description is a list of related routines. The user is advised to look at these as the list may include routines as the list may include routines more suited for the application being considered. Conversely the routines may shed further light on how the original routine should be used.

The descriptions of the routines are for the default routine that the entry jumps to. The user may change the entry and this may alter the action of the routine. The user is advised to stick to the entry/exit conditions described otherwise programs that call the routine (BASIC for example) may cease to operate correctly.

Page 14.1 → 17.21
APP1.1 → APP12-8

0: KM INITIALISE #BB00

Initialise the Key Manager.

Action:

Full initialisation of the Key Manager (as used during EMS). All Key Manager variables, buffers and indirections are initialised. The previous state of the Key Manager is lost.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The Key Manager indirection (KM TEST BREAK) is set to its default routine.
The key buffer is set up (to be empty).
The expansion buffer is set up and the expansions are set to their default strings.
The key translation tables are initialised to their default translations.
The repeating key map is initialised to its default state.
The repeat speeds are set to their default values.
Shift and caps lock are turned off.
The break event is disarmed.

See Appendices II, III and IV for the default translation tables, repeating key table and expansion strings.

This routine enables interrupts.

Related entries:

KM RESET

1: KM RESET #BB03

Reset the Key Manager.

Action:

Re-initialises the Key Manager indirections and buffers.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The Key Manager indirection (KM TEST BREAK) is set to its default routine.
The key buffer is set up (to be empty).
The expansion buffer is set up and the expansions are set to their default strings (see Appendix IV).
The break event is disarmed.

All pending keys and characters are discarded.

This routine enables interrupts.

Related entries:

KM DISARM BREAK
KM EXP BUFFER
KM INITIALISE

2: KM WAIT CHAR #BB06

Wait for next character from the keyboard.

Action:

Try to get a character from the key buffer or the current expansion string. This routine waits until a character is available if no character is immediately available.

Entry conditions:

No conditions.

Exit conditions:

Carry true.
A contains the character.

Other flags corrupt.
All other registers preserved.

Notes:

The possible sources for generating the next character are, in the order that they are tested:

The 'put back' character.
The next character of an expansion string.
The first character of an expansion string.
A character from a key translation table.

Expansion tokens found in the key translation table are expanded to their associated strings. Expansion tokens found in expansion strings are not expanded but are treated as characters.

Related entries:

KM CHAR RETURN
KM READ CHAR
KM WAIT KEY

3: KM READ CHAR #BB09

Test if a character is available from the keyboard.

Action:

Try to get a character from the key buffer or the current expansion string. This routine does not wait for a character to become available if there is no character available immediately.

Entry conditions:

No conditions.

Exit conditions:

If there was a character available:

Carry true.
A contains the character.

If there was no character available:

Carry false.
A corrupt.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The possible sources for generating the next character are, in the order that they are tested:

The 'put back' character.
The next character of an expansion string.
The first character of an expansion string.
A character from a key translation table.

Expansion tokens in the key translation tables will be expanded to their associated strings. Expansion tokens found in expansion strings are not expanded but are treated as characters.

This routine will always return a character if one is available. It is therefore possible to flush out the Key Manager buffers by calling KM READ CHAR repeatedly until it reports that no character is available.

Related entries:

KM CHAR RETURN
KM READ KEY
KM WAIT CHAR

4: KM CHAR RETURN #BB0C

Return a single character to the keyboard for next time.

Action:

Save a character for the next call of KM READ CHAR or KM WAIT CHAR.

Entry conditions:

A contains the character to put back.

Exit conditions:

All registers and flags preserved.

Notes:

The 'put back' character will be returned before any other character is generated by the keyboard. It will not be expanded (or otherwise dealt with) but will be returned as it is. The 'put back' character need not have been read from the keyboard, it could be inserted by the user for some purpose.

It is only possible to have one 'put back' character. If this routine is called twice without reading a character between these then the first 'put back' will be lost. Furthermore, it is not possible to return character 255 (because this is used as the marker for no 'put back' character).

Related entries:

KM READ CHAR
KM WAIT CHAR

5: KM SET EXPAND #BB0F

Set an expansion string.

Action:

Set the expansion string associated with an expansion token.

Entry conditions:

B contains the expansion token for the expansion to set.
C contains the length of the string.
HL contains the address of the string.

Exit conditions:

If the expansion is OK:

Carry true.

If the string was too long or the token was invalid:

Carry false.

Always:

A, BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

The string to be set may lie anywhere in RAM. Expansion strings cannot be set directly from ROM.

The characters in the string are not expanded (or otherwise dealt with). It is therefore possible to put any character into an expansion string.

If there is insufficient room in the expansion buffer for the new string then no change is made to the expansions.

If the string set is currently being used to generate characters (by KM READ CHAR or KM WAIT CHAR) then the unread portion of the string is discarded. The next character will be read from the key buffer.

This routine enables interrupts.

Related entries:

KM GET EXPAND
KM READ CHAR
KM WAIT CHAR

6: KM GET EXPAND #BB12

Get a character from an expansion string.

Action:

Read a character from an expansion string. The characters in the string are numbered starting from 0.

Entry conditions:

A contains an expansion token.
L contains the character number.

Exit conditions:

If the character was found:

Carry true.
A contains the character.

If the token was invalid or the string was not long enough:

Carry false.
A corrupt.

Always:

DE and other flags corrupt.
All other registers preserved.

Notes:

The characters in the expansion string are not expanded (or otherwise dealt with). It is therefore possible to put any character into an expansion string.

Related entries:

KM READ CHAR
KM SET EXPAND

7: KM EXP BUFFER #BB15

Allocate a buffer for expansion strings.

Action:

Set the address and length of the expansion buffer. Initialise the buffer with the default expansion strings.

Entry conditions:

DE contains the address of the buffer.
HL contains the length of the buffer.

Exit conditions:

If the buffer is OK:

Carry true.

If the buffer is too short:

Carry false.

Always:

A, BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

The buffer must not be located underneath a ROM and it must be at least 49 bytes long (i.e. have sufficient space for the default expansion strings). If the new buffer is too short then the old buffer is left unchanged.

The default expansion strings are given in Appendix IV.

Any expansion string currently being read is discarded.

This routine enables interrupts.

Related entries.

KM GET EXPAND
KM SET EXPAND

8: KM WAIT KEY #BB18

Wait for next key from the keyboard.

Action:

Try to get a key from the key buffer. This routine waits until a key is found if no key is immediately available.

Entry conditions:

No conditions.

Exit conditions:

Carry true.
A contains the character or expansion token.

Other flags corrupt.
All other registers preserved.

Notes:

The next key is read from the key buffer and translated using the appropriate key translation table. Expansion tokens are not expanded but are passed out for the user to deal with, as are normal characters. Other Key Manager tokens (shift lock, caps lock and ignore) are obeyed but are not passed out.

Related entries:

KM READ KEY
KM WAIT CHAR

9: KM READ KEY #BB1B

Test if a key is available from the keyboard.

Action:

Try to get a key from the key buffer. This routine does not wait if no key is available immediately.

Entry conditions:

No conditions.

Exit conditions:

If a key was available:

Carry true.
A contains the character or expansion token.

If no key was available:

Carry false.
A corrupt.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The next key is read from the key buffer and translated using the appropriate key translation table. Expansion tokens are not expanded but are passed out for the user to deal with, as are normal characters. Other Key Manager tokens (shift lock, caps lock and ignore) are obeyed but are not passed out.

This routine will always return a key if one is available. It is therefore possible to flush out the key buffer by calling KM READ KEY repeatedly until it claims no key is available. Note, however, that the 'put back' character or a partially read expansion string is ignored. It is advisable to use KM READ CHAR to flush these out when emptying the Key Manager buffers.

Related entries:

KM READ CHAR
KM WAIT KEY

10: KM TEST KEY #BB1E

Test if a key is pressed.

Action:

Test if a particular key or joystick button is pressed. This is done using the key state map rather than by accessing the keyboard hardware.

Entry conditions:

A contains a key number.

Exit conditions:

If the key is pressed:

Zero false.

If the key is not pressed:

Zero true.

Always:

Carry false.
C contains the current shift and control state.

A, HL and other flags corrupt.
All other registers preserved.

Notes:

The shift and control states are automatically read when a key is scanned. If bit 7 is set then the control key is pressed and if bit 5 is set then one of the shift keys is pressed.

The key number is not checked. An invalid key number will generate the correct shift and control states but the state of the key tested will be meaningless.

The key state map which this routine tests is updated by the keyboard scanning routine. Normally this is run every fiftieth of a second and so the state may be out of date by that much. The key debouncing requires that a key should be released for two scans of the keyboard before it is marked as released in the key state map; the pressing of a key is detected immediately.

Related entries:

KM GET JOYSTICK
KM GET STATE
KM READ KEY

11: KM GET STATE #BB21

Fetch Caps Lock and Shift Lock states.

Action:

Ask if the keyboard is currently shift locked or caps locked.

Entry conditions:

No conditions.

Exit conditions:

L contains the shift lock state.
H contains the caps lock state.

AF corrupt.
All other registers preserved.

Notes:

The lock states are:

#00 means the lock is off
#FF means the lock is on

The default lock states are off.

Related entries:

KM TEST KEY

12: KM GET JOYSTICK #BB24

Fetch current state of the joystick(s).

Action:

Ask what the current states of the joysticks are. These are read from the key state map rather than by accessing the keyboard hardware.

Entry conditions:

No conditions.

Exit conditions:

H contains the state of joystick 0.
L contains the state of joystick 1.
A contains the state of joystick 0.

Flags corrupt.
All other registers preserved.

Notes:

In normal operation the key state map is updated by the key scanning routine every fiftieth of a second so the state returned may be slightly out of date.

The joystick states are bit significant as follows:

Bit 0	Up.
Bit 1	Down.
Bit 2	Left.
Bit 3	Right.
Bit 4	Fire 2.
Bit 5	Fire 1.
Bit 6	Spare joystick button (usually unconnected).
Bit 7	Always zero.

If a bit is set then the appropriate button is pressed.

Joystick 1 is indistinguishable from certain keys on the keyboard (see Appendix I).

Related entries:

KM TEST KEY

13: KM SET TRANSLATE #BB27

Set entry in normal key translation table

Action:

Set what character or token a key will be translated to when neither shift nor control is pressed.

Entry conditions:

A contains a key number.
B contains the new translation.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET TRANSLATE
KM SET CONTROL
KM SET SHIFT

14: KM GET TRANSLATE #BB2A

Get entry from normal key translation table.

Action:

Ask what character or token a key will be translated to when neither shift nor control is pressed.

Entry conditions:

A contains a key number.

Exit conditions:

A contains the current translation.

HL and flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the translation returned is meaningless.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET SHIFT
KM SET TRANSLATE

15: KM SET SHIFT #BB2D

Set entry in shifted key translation table

Action:

Set what character or token a key will be translated to when control is not pressed but shift is pressed or shift lock is on.

Entry conditions:

A contains a key number.
B contains the new translation.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET SHIFT
KM SET CONTROL
KM SET TRANSLATE

16: KM GET SHIFT #BB30

Get entry from shifted key translation table.

Action:

Ask what character or token a key will be translated to when control is not pressed but shift is pressed or shift lock is on.

Entry conditions:

A contains a key number.

Exit conditions:

A contains the current translation.

HL and flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the translation returned is meaningless.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM GET TRANSLATE
KM SET SHIFT

17: KM SET CONTROL #BB33

Set entry in control key translation table

Action:

Set what character or token a key will be translated to when control is pressed.

Entry conditions:

A contains a key number.
B contains the new translation.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET CONTROL
KM SET SHIFT
KM SET TRANSLATE

18: KM GET CONTROL #BB36

Get entry from control key translation table

Action:

Ask what character or token a key will be translated to when control is pressed.

Entry conditions:

A contains a key number.

Exit conditions:

A contains the current translation.
HL and flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the translation returned is meaningless.

Most values in the table are treated as characters and are passed back to the user. However, there are certain special values:

#80..#9F	are the expansion tokens and are expanded to character strings when KM READ CHAR or KM WAIT CHAR is called although they are passed back like any other character when KM READ KEY or KM WAIT KEY is called.
#FD	is the caps lock token and causes the caps lock to toggle (turn on if off and vice versa).
#FE	is the shift lock token and causes the shift lock to toggle (turn on if off and vice versa).
#FF	is the ignore token and means the key should be thrown away.

Characters #E0..#FC have special meanings to the BASIC to do with editing, cursoring and breaks.

See Appendix II for a full listing of the default translation tables.

Related entries:

KM GET SHIFT
KM GET TRANSLATE
KM SET CONTROL

19: KM SET REPEAT #BB39

Set whether a key may repeat.

Action:

Set the entry in the repeating key map that determines whether a key is allowed to repeat or not.

Entry conditions:

If the key is to be allowed to repeat:

B contains #FF.

If the key is not to be allowed to repeat:

B contains #00.

Always:

A contains the key number.

Exit conditions:

AF, BC and HL corrupt.
All other registers preserved.

Notes:

If the key number is invalid (greater than 79) then no action is taken.

The default repeating keys are listed in Appendix III.

Related entries:

KM GET REPEAT
KM SET DELAY

20: KM GET REPEAT #BB3C

Ask if a key is allowed to repeat.

Action:

Test the entry in the repeating key map that says whether a key is allowed to repeat or not.

Entry conditions:

A contains a key number.

Exit conditions:

If the key is allowed to repeat:

Zero false.

If the key is not allowed to repeat:

Zero true.

Always:

Carry false.
A, HL and other flags corrupt.
All other registers preserved.

Notes:

The key number is not checked. If it is invalid (greater than 79) then the repeat state returned is meaningless.

The default repeating keys are listed in Appendix III.

Related entries:

KM SET REPEAT

21: KM SET DELAY #BB3F

Set start up delay and repeat speed.

Action:

Set the time before keys first repeat (start up delay) and the time between repeats (repeat speed).

Entry conditions:

H contains the new start up delay.
L contains the new repeat speed.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

Both delays are given in scans of the keyboard. The keyboard is scanned every fiftieth of a second.

A start up delay or repeat speed of 0 is taken to mean 256.

The default start up delay is 30 scans (0.6 seconds) and the default repeat speed is 2 scans (0.04 seconds or 25 characters a second).

Note that a key is prevented from repeating (by the key scanner) if the key buffer is not empty. Thus the actual repeat speed is the slower of the supplied repeat speed and the rate at which characters are removed from the buffer. This is intended to prevent the user from getting too far ahead of a program that is running sluggishly.

The start up delay and repeat speed apply to all keys on the keyboard that are set to repeat.

Related entries:

KM GET DELAY
KM SET REPEAT

22: KM GET DELAY #BB42

Get start up delay and repeat speed.

Action:

Ask the time before keys first repeat (start up delay) and the time between repeats (repeat speed).

Entry conditions:

No conditions.

Exit conditions:

H contains the start up delay.
L contains the repeat speed.

AF corrupt.
All other registers preserved.

Notes:

Both delays are given in scans of the keyboard. The keyboard is scanned every fiftieth of a second.

A repeat speed or start up delay of 0 means 256.

Related entries:

KM SET DELAY

23: KM ARM BREAKS #BB45

Allow break events to be generated.

Action:

Arm the break mechanism. The next call of KM BREAK EVENT will generate a break event.

Entry conditions:

DE contains the address of the break event routine.
C contains the ROM select address for this routine.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The break mechanism can be disarmed by calling KM DISARM BREAK (or KM RESET).

This routine enables interrupts.

Related entries:

KM BREAK EVENT
KM DISARM BREAK

24: KM DISARM BREAK #BB48

Prevent break events from being generated.

Action:

Disarm the break mechanism. From now on the generation of break events by KM BREAK EVENT will be suppressed.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

Break events can be rearmed by calling KM ARM BREAK.

The default state of the break mechanism is disarmed, thus calling KM RESET will also disarm breaks.

This routine enables interrupts.

Related entries:

KM ARM BREAK
KM BREAK EVENT

25: KM BREAK EVENT #BB4B

Generate a break event (if armed).

Action:

Try to generate a break event.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

If the break mechanism is disarmed then no action is taken. Otherwise a break event is generated and a special marker is placed into the key buffer. This marker generates a break event token (#EF) when read from the buffer. The break mechanism is automatically disarmed after generating a break event so that multiple breaks can be avoided.

This routine may be run from the interrupt path and thus does not and should not enable interrupts. Note, however, that using a LOW JUMP to call the routine (as the firmware jumpblock is set up to do) does enable interrupts and so the jumpblock may not be used directly from interrupt routines.

Related entries:

KM ARM BREAK
KM DISARM BREAK

26: TXT INITIALISE #BB4E

Initialise the Text VDU.

Action:

Full initialisation of the Text VDU (as used during EMS). All Text VDU variables and indirections are initialised, the previous VDU state is lost.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The Text VDU indirections (TXT DRAW CURSOR, TXT UNDRAW CURSOR, TXT WRITE CHAR, TXT UNWRITE and TXT OUT ACTION) are set to their default routines.

The control code table is set up to perform the default control code actions.

The user defined character table is set to be empty.
Stream 0 is selected.

All streams are set to their default states:

> The text paper (background) is set to ink 0.
> The text pen (foreground) is set to ink 1.
> The text window is set to the entire screen.
> The text cursor is enabled but turned off.
> The character writing mode is set to opaque.
> The VDU is enabled.
> The graphic character write mode is turned off.
> The cursor is moved to the top left corner of the window.

The default character set and the default setting for the control code table are described in Appendices VI and VII.

Related Entries:

SCR INITIALISE
TXT RESET

27: TXT RESET #BB51

Reset the Text VDU.

Action:

Reinitialises the Text VDU indirections and the control code table. Does not affect any other aspect of the Text VDU.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The Text VDU indirections TXT DRAW CURSOR, TXT UNDRAW CURSOR, TXT WRITE CHAR, TXT UNWRITE and TXT OUT ACTION are set to their default routines.

The control code table is set up to perform the default control code actions (see Appendix VII).

Related Entries:

TXT INITIALISE

28: TXT VDU ENABLE #BB54

Allow characters to be placed on the screen.

Action:

Permit characters to be printed when requested (by calling TXT OUTPUT or TXT WR CHAR). Enabling applies to the currently selected stream. The cursor blob is also enabled (by calling TXT CUR ENABLE).

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The control code buffer used by TXT OUTPUT is emptied, any incomplete control code sequence will be lost.

Related Entries:

TXT CUR ENABLE
TXT OUTPUT
TXT VDU DISABLE
TXT WR CHAR

29: TXT VDU DISABLE #BB57

Prevent characters being placed on the screen.

Action:

Prevents characters being printed on the screen (when TXT OUTPUT or TXT WR CHAR is called). Applies to the currently selected stream. The cursor blob is also disabled (by calling TXT CUR DISABLE).

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The control code buffer used by TXT OUTPUT is emptied, any incomplete control sequence will be lost.

Control codes are still obeyed by TXT OUTPUT.

Related Entries:

TXT CUR DISABLE
TXT OUTPUT
TXT VDU ENABLE
TXT WR CHAR

30: TXT OUTPUT #BB5A

Output a character or control code to the Text VDU.

Action:

Output characters to the screen and obey control codes (characters #00..#1F). Works on the currently selected stream.

Entry conditions:

A contains the character to send.

Exit conditions:

All registers and flags preserved.

Notes:

This routine calls the TXT OUT ACTION indirection to do the work of printing the character or obeying the control code described below.

Control codes may take up to 9 parameters. These are the characters sent following the initial control code. The characters sent are stored in the control code buffer until sufficient have been received to make up all the required parameters. The control code buffer is only long enough to accept 9 parameter characters.

There is only one control code buffer for all streams. It is therefore possible to get unpredictable results if the output stream is changed midway through sending a control code sequence.

If the VDU is disabled then no characters will be printed on the screen. Control codes will still be obeyed, however, the user should avoid using this 'facility' where possible.

If the graphic character write mode is enabled then all characters and control codes are printed using the Graphics VDU routine, GRA WR CHAR, and are not obeyed.

Characters are written in the same way that TXT WR CHAR writes characters.

Related Entries:

GRA WR CHAR
TXT OUT ACTION
TXT SET GRAPHIC
TXT VDU DISABLE
TXT VDU ENABLE
TXT WR CHAR

31: TXT WR CHAR #BB5D

Write a character to the screen.

Action:

Print a character on the screen at the cursor position of the currently selected stream. Control codes (characters #00..#1F) are printed and not obeyed.

Entry conditions:

A contains the character to print.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

If the VDU is disabled then no character will be printed.

Before printing the character the cursor position is forced to lie within the text window (see TXT VALIDATE). After printing the character the cursor is moved right one character.

To put the character on the screen this routine calls the TXT WRITE CHAR indirection.

Related Entries:

GRA WR CHAR
TXT OUTPUT
TXT RD CHAR
TXT WRITE CHAR

32: TXT RD CHAR #BB60

Read a character from the screen.

Action:

Read a character from the screen at the cursor position of the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

If a recognisable character was found:

Carry true.
A contains the character read.

If no recognisable character was found:

Carry false.
A contains zero.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The cursor position is not forced legal (inside the window) before the character is read. Steps must be taken to avoid reading characters from outside the text window (or even off the screen)!

The read is performed by comparing the matrix found on the screen with the matrices used to generate characters. As a result changing a character matrix, changing the pen or paper inks, or changing the screen (e.g. drawing a line through a character) may make the character unreadable.

To actually read the character from the screen the TXT UNWRITE indirection is called.

Special precautions are taken against space being generated. Initially the character is read assuming that the character was written in the current pen ink and treating any other ink as background. If this fails to generate a recognisable character or it generates space then another try is made by assuming that the background to the character was written in the current paper ink and treating any other ink as foreground.

The characters are scanned starting with #00 and finishing with #FF.

Related Entries:

TXT UNWRITE
TXT WR CHAR

33: TXT SET GRAPHIC #BB63

Turn on or off the Graphics VDU write character option.

Action:

Enable or disable graphic character writing on the currently selected stream.

Entry conditions:

If graphic writing is to be turned on:

A must be non-zero.

If graphic writing is to be turned off:

A must contain zero.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

When graphic character writing is enabled then all characters sent to TXT OUTPUT are printed using the Graphics VDU (see GRA WR CHAR) rather than the Text VDU (see TXT WR CHAR). Also all control codes are printed rather than obeyed. Characters sent to TXT WR CHAR will be printed as normal.

Character printing is not prevented by disabling the Text VDU (with TXT VDU DISABLE) if graphic character writing is enabled.

Related Entries:

GRA WR CHAR
TXT OUTPUT

34: TXT WIN ENABLE #BB66

Set the size of the current text window.

Action:

Set the boundaries of the window on the currently selected stream. The edges are the first and last character columns inside the window and the first and last character rows inside the window.

Entry conditions:

H contains the physical column of one edge.
D contains the physical column of the other edge.
L contains the physical row of one edge.
E contains the physical row of the other edge.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The edge positions are given in physical screen coordinates. i.e. Row 0, column 0 is the top left corner of the screen and the coordinates are signed numbers.

The window is truncated, if necessary, so that it fits on the screen.

The left column of the window is taken to be the smaller of H and D. The top row of the window is taken to be the smaller of L and E.

The cursor is moved to the top left corner of the window.

The window is not cleared.

If the window covers the whole screen then when the window is rolled the hardware roll routine (see SCR HW ROLL) will be used. If the window covers less than the whole screen the software roll routine (see SCR SW ROLL) will be used.

The default text window covers the whole screen and is set up when TXT INITIALISE or SCR SET MODE is called.

Related Entries:

TXT GET WINDOW
TXT VALIDATE

35: TXT GET WINDOW #BB69

Get the size of the current window.

Action:

Get the boundaries of the window on the currently selected stream and whether it covers the whole screen.

Entry conditions:

No conditions.

Exit conditions:

If the window covers the whole screen:

Carry false.

If the window covers less than the whole screen:

Carry true.

Always:

H contains the leftmost column in the window.
D contains the rightmost column in the window.
L contains the topmost row in the window.
E contains the bottommost row in the window.

A corrupt.
All other registers preserved.

Notes:

The boundaries of the window are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen.

The boundaries returned by this routine may not be the same as those set when TXT WIN ENABLE was called because the window is truncated to fit the screen.

Related Entries:

TXT VALIDATE
TXT WIN ENABLE

36: TXT CLEAR WINDOW #BB6C

Clear current window.

Action:

Clear the text window of the currently selected stream to the paper ink of the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The cursor is moved to the top left corner of the window.

Related Entries:

GRA CLEAR WINDOW
SCR CLEAR
TXT SET PAPER
TXT WIN ENABLE

37: TXT SET COLUMN #BB6F

Set cursor horizontal position.

Action:

Move the current position of the currently selected stream to a new column. The cursor blob will be removed from the current position and redrawn at the new position (if the cursor is enabled and turned on).

Entry conditions:

A contains the required logical column for the cursor.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The required column is given in logical coordinates. i.e. Column 1 is the leftmost column of the window.

The cursor may be moved outside the window. However, it will be forced to lie inside the window before any character is written by the Text VDU (see TXT VALIDATE) or the cursor blob is drawn.

Related Entries:

TXT GET CURSOR
TXT SET CURSOR
TXT SET ROW

38: TXT SET ROW #BB72

Set cursor vertical position.

Action:

Move the current position of the currently selected stream to a new row. The cursor blob will be removed from the current position and redrawn at the new position (if the cursor is enabled and turned on).

Entry conditions:

A contains the required logical row for the cursor.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The required row is given in logical coordinates. i.e. Row 1 is the topmost row of the window.

The cursor may be moved outside the window. However, it will be forced to lie inside the window before any character is written by the Text VDU (see TXT VALIDATE) or the cursor blob is drawn.

Related Entries:

TXT GET CURSOR
TXT SET COLUMN
TXT SET CURSOR

39: TXT SET CURSOR #BB75

Set cursor position.

Action:

Move the current position of the currently selected stream to a new row and column. The cursor blob will be removed from the current position and redrawn at the new position (if the cursor is enabled and turned on).

Entry conditions:

H contains the required logical column.
L contains the required logical row.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The required position is given in logical coordinates. i.e. Row 1, column 1 is the top left corner of the window.

The cursor position may be moved outside the window. However, it will be forced to lie inside the window before any character is written by the Text VDU (see TXT VALIDATE) or the cursor blob is drawn.

Related Entries:

TXT GET CURSOR
TXT SET COLUMN
TXT SET ROW

40: TXT GET CURSOR #BB78

Ask current cursor position.

Action:

Get the current location of the cursor and a count of the number of times the window of the currently selected stream has rolled.

Entry conditions:

No conditions.

Exit conditions:

H contains the logical cursor column.
L contains the logical cursor row.
A contains the current roll count.

Flags corrupt.
All other registers are preserved.

Notes:

The cursor position is given in logical coordinates. i.e. Row 1, column 1 is the top left corner of the window.

The roll count passed out has no absolute meaning. It is decremented when the window is rolled up and is incremented when the window is rolled down. It may be used to determine whether the window has rolled by comparing it with a previous value.

The position reported may not be inside the window and is, therefore, not necessarily the position at which the next character will be printed. Use TXT VALIDATE to check this.

Related Entries:

TXT SET COLUMN
TXT SET CURSOR
TXT SET ROW
TXT VALIDATE

41: TXT CUR ENABLE #BB7B

Allow cursor display - user.

Action:

Allow the cursor blob for the currently selected stream to be placed on the screen. The cursor blob will be placed on the screen immediately unless the cursor is turned off (see TXT CUR OFF).

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

Cursor enabling and disabling is intended for use by the user. It is also used when the VDU is disabled (see TXT VDU ENABLE and TXT VDU DISABLE).

Related Entries:

TXT CUR DISABLE
TXT CUR ON
TXT DRAW CURSOR
TXT UNDRAW CURSOR

42: TXT CUR DISABLE #BB7E

Disallow cursor display - user.

Action:

Prevent the cursor blob for the currently selected stream from being placed on the screen. The cursor blob will be removed from the screen immediately if it is currently there.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

Cursor enabling and disabling is intended for use by the user. It is also used when the VDU is disabled (see TXT VDU ENABLE and TXT VDU DISABLE).

Related Entries:

TXT CUR ENABLE
TXT CUR OFF
TXT DRAW CURSOR
TXT UNDRAW CURSOR

43: TXT CUR ON #BB81

Allow cursor display - system.

Action:

Allow the cursor blob for the currently selected stream to be placed on the screen. The cursor blob will be placed on the screen immediately unless the cursor is disabled (see TXT CUR DISABLE).

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

Turning the cursor on and off is intended for use by system ROMs.

Related Entries:

TXT CUR ENABLE
TXT CUR OFF
TXT DRAW CURSOR
TXT UNDRAW CURSOR

44: TXT CUR OFF #BB84

Disallow cursor display - system.

Action:

Prevent the cursor blob for the currently selected stream from being placed on the screen. The cursor blob will be removed from the screen immediately if it is currently there.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

Turning the cursor on and off is intended for use by system ROMs.

Related Entries:

TXT CUR DISABLE
TXT CUR ON
TXT DRAW CURSOR
TXT UNDRAW CURSOR

45: TXT VALIDATE #BB87

Check if a cursor position is within the window.

Action:

Check a screen position to see if it lies within the current window. If it does not then determine the position where a character would be printed after applying the rules for forcing the screen position inside the window.

Entry conditions:

H contains the logical column of the position to check.
L contains the logical row of the position to check.

Exit conditions:

If printing at the position would not cause the window to roll:

> Carry true.
> B corrupt.

If printing at the position would cause the window to roll up:

> Carry false.
> B contains #FF.

If printing at the position would cause the window to roll down:

> Carry false.
> B contains #00.

Always:

> H contains the logical column at which a character would be printed.
> L contains the logical row at which a character would be printed.
>
> A and other flags corrupt.
> All other registers preserved.

Notes:

The positions on the screen are given in logical coordinates. i.e. Row 1, column 1 is the top left corner of the window.

Before writing a character or putting the cursor blob on the screen the Text VDU validates the current position, performs any required roll then writes at the appropriate position.

The algorithm to work out the position to print at, from the position to check, is as follows:

1/ If the position is right of the right edge of the window it is moved to the left edge of the window on the next line.

2/ If the position is left of the left edge of the window it is moved to the right edge of the window on the previous line.

3/ If the position is now above the top edge of the window then it is moved to the top edge of the window and the window needs rolling downwards.

4/ If the position is now below the bottom edge of the window it is moved to the bottom edge of the window and the window needs rolling upwards.

Related Entries:

SCR HW ROLL
SCR SW ROLL
TXT GET CURSOR

46: TXT PLACE CURSOR #BB8A

Put a cursor blob on the screen.

Action:

Put a cursor blob on the screen at the cursor position for the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

TXT PLACE CURSOR is provided to allow the user to run multiple cursors in a window. The indirection TXT DRAW CURSOR should be called for merely placing the normal cursor blob on the screen. Higher level routines, such as TXT OUTPUT and TXT SET CURSOR, automatically remove and place the normal cursor when appropriate, the user must deal with any other cursors.

It is not safe to call TXT PLACE CURSOR twice at a particular screen position without calling TXT REMOVE CURSOR in between because this may leave a spurious cursor blob on the screen when the cursor position is moved.

The cursor position is forced to be inside the window before the cursor blob is drawn.

The cursor blob is an inverse patch formed by exclusive-oring the contents of the screen at the cursor position with the exclusive-or of the current pen and paper inks.

Related Entries:

TXT DRAW CURSOR
TXT REMOVE CURSOR

47: TXT REMOVE CURSOR #BB8D

Take a cursor blob off the screen.

Action:

Take a cursor blob off the screen at the cursor position of the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

TXT REMOVE CURSOR is provided to allow the user to run multiple cursors in a window. The indirection TXT UNDRAW CURSOR should be called for merely removing the normal cursor from the screen. Higher level routines, such as TXT OUTPUT and TXT SET CURSOR, automatically remove and place the normal cursor when appropriate, the user must deal with any other cursors.

TXT REMOVE CURSOR should only be used to remove a cursor placed on the screen by calling TXT PLACE CURSOR. The cursor should be removed when the cursor position is to be changed (rolling the window implicitly changes the cursor position) or the screen is to be read or written. Incorrect use of this routine may result in a spurious cursor blob being generated.

The cursor position is forced to be inside the window before the cursor blob is removed (this should not matter as TXT PLACE CURSOR has already done this).

The cursor blob is an inverse patch formed by exclusive-oring the contents of the screen at the cursor position with the exclusive-or of the current pen and paper inks.

Related Entries:

TXT PLACE CURSOR
TXT UNDRAW CURSOR

48: TXT SET PEN #BB90

Set ink for writing characters.

Action:

Set the text pen ink for the currently selected stream. This is the ink that is used for writing characters (the foreground ink).

Entry conditions:

A contains ink to use.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The ink is masked to bring it within the range of legal inks for the current screen mode. That is with #0F in mode 0, #03 in mode 1 and #01 in mode 2.

The cursor blob will be redrawn using the new ink (if enabled).

Related Entries:

GRA SET PEN
SCR SET INK
TXT GET PEN
TXT SET PAPER

49: TXT GET PEN #BB93

Get ink for writing characters.

Action:

Ask what the pen ink is set to for the currently selected stream. This is the ink used for writing characters (foreground ink).

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related Entries:

GRA GET PEN
SCR GET INK
TXT GET PAPER
TXT SET PEN

50: TXT SET PAPER #BB96

Set ink for writing text background.

Action:

Set the text paper ink for the currently selected stream. This is the ink used for writing the background to characters and for clearing the text window.

Entry conditions:

A contains the ink to use.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The ink is masked to bring it within the range of legal inks for the current screen mode. That is with #0F in mode 0, #03 in mode 1 and #01 in mode 2.

The cursor blob will be redrawn using the new ink (if enabled).

This ink will be used when clearing areas of the text window (by TXT CLEAR WINDOW and certain control codes).

This routine does not clear the text window.

Related Entries:

GRA SET PAPER
SCR SET INK
TXT GET PAPER
TXT SET PEN

51: TXT GET PAPER #BB99

Get ink for writing background.

Action:

Ask what the paper ink is set to for the currently selected stream. This is the ink used for writing the background to characters and for clearing the text window.

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related Entries:

GRA GET PAPER
SCR GET INK
TXT GET PEN
TXT SET PAPER

52: TXT INVERSE #BB9C

Swap current pen and paper inks over.

Action:

Exchange the text pen and paper (foreground and background) inks for the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The cursor blob is not redrawn and so it should not be on the screen when this routine is called.

Related Entries:

TXT SET PAPER
TXT SET PEN

53: TXT SET BACK #BB9F

Allow or disallow background being written.

Action:

Set character write mode to opaque or transparent for the currently selected stream. Opaque mode writes background with the character. Transparent mode writes the character on top of the current contents of the screen.

Entry conditions:

If background is to be written (opaque mode):

A must be zero.

If background is not to be written (transparent mode):

A must be non-zero.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

Writing in transparent mode is intended for annotating diagrams and similar applications. It can have unfortunate effects if it is used generally because overwriting a character will not remove the character underneath thus creating an incomprehensible jumble on the screen.

Setting transparent mode does not affect the Graphics VDU because GRA WR CHAR always prints opaque.

Related Entries:

TXT GET BACK
TXT WR CHAR
TXT WRITE CHAR

54: TXT GET BACK #BBA2

Ask if background is being written.

Action:

Get the character write mode for the currently selected stream.

Entry conditions:

No conditions.

Exit conditions:

If background is to be written (opaque mode):

A contains zero.

If background is not to be written (transparent mode):

A contains non-zero.

Always:

DE, HL and flags corrupt.
All other registers preserved.

Notes:

This only applies to the Text VDU, the Graphics VDU always writes opaque.

Related Entries:

TXT SET BACK

55: TXT GET MATRIX #BBA5

Get the address of a character matrix.

Action:

Calculate a pointer to the matrix for a character and determine if it is a user defined matrix.

Entry conditions:

A contains the character whose matrix is to be found.

Exit conditions:

If the matrix in the user defined matrix table:

Carry true.

If the matrix is in the lower ROM:

Carry false.

Always:

HL contains the address of the matrix.

A and other flags corrupt.
All other registers preserved.

Notes:

The matrix may be in RAM or in ROM. The Text VDU assumes that the appropriate ROMs are enabled or disabled when it calls this routine to get the matrix for a character. (The lower ROM is on, the upper ROM is normally off).

The matrix is stored as an 8 byte bit significant vector. The first byte describes the top line of the character and the last byte the bottom line. Bit 7 of a byte refers to the leftmost pixel of a line and bit 0 to the rightmost pixel. If a bit is set in the matrix then the pixel should be written in the pen ink. If a bit is not set then the pixel should either be written in the paper ink or left alone (depending on the opaque/transparent write mode).

Related Entries:

TXT SET MATRIX

56: TXT SET MATRIX #BBA8

Set a character matrix.

Action:

Set the matrix for a user defined character. If the character is not user defined then no action is taken.

Entry conditions:

A contains the character whose matrix is to be set.
HL contains the address of the matrix to set.

Exit conditions:

If the character is user definable:

Carry true.

If the character is not user definable:

Carry false.

Always:

A, BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

The matrix is stored as an 8 byte bit significant vector. The first byte describes the top line of the character and the last byte the bottom line. Bit 7 of a byte refers to the leftmost pixel of a line and bit 0 to the rightmost pixel. If a bit is set in the matrix then the pixel should be written in the pen ink. If a bit is not set then the pixel should either be written in the paper ink or left alone (depending whether the character write mode is opaque or transparent currently).

The matrix is copied from the area given into the character matrix table without using RAM LAMs thus the matrices can be set from ROM providing it is enabled. (Note however that the jumpblock disables the upper ROM.)

Altering a character matrix changes the matrix for all streams. It does not alter any character on the screen; it changes what will be placed on the screen the next time the character is written.

Related Entries:

TXT GET MATRIX
TXT SET M TABLE

57: TXT SET M TABLE #BBAB

Set the user defined matrix table address.

Action:

Set the user defined matrix table and the number of characters in the table. The table is initialised with the current matrix settings.

Entry conditions:

DE contains the first character in the table.
HL contains the address of the start of the new table.

Exit conditions:

If there was no user defined matrix table before:

> Carry false.
> A and HL corrupt.

If there was a user defined matrix table before:

> Carry true.
> A contains the first character in the old table.
> HL contains the address of the old table.

Always:

> BC, DE and other flags corrupt.
> All other registers preserved.

Notes:

If the first character specified is in the range 0..255 then the matrices for all characters between that character and character 255 are to be stored in the user defined table.

If the first character specified is not in the range 0..255 then the user defined matrix table is deemed to contain no matrices (and the table address passed is ignored).

The table must be (256 – first char) * 8 bytes long. The matrices are stored in the table in ascending order. The table is initialised with the current matrix settings, whether they were previously in RAM or in the ROM.

The table should not be located in RAM underneath a ROM.

It is permissible for the new and old matrix tables to overlap (thus allowing the table to be extended or contracted) providing that matrices in the new table occupy an earlier or equal address than they occupied in the old table.

All streams share the matrix table so any changes to it will be reflected on all streams.

Related Entries:

TXT GET M TABLE
TXT SET MATRIX

58: TXT GET M TABLE #BBAE

Get user defined matrix table address.

Action:

Get the address of the current user defined matrix table and the first character in the table.

Entry conditions:

No conditions.

Exit conditions:

If there is no user defined matrix table:

Carry false.
A and HL corrupt.

If there is a user defined matrix table:

Carry true.
A contains the first character in the table.
HL contains the address of the start of the table.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The matrices for characters between the first character and 255 are stored in the table in ascending order. Each matrix is 8 bytes long.

Related Entries:

TXT GET MATRIX
TXT SET M TABLE

59: TXT GET CONTROLS #BBB1

Fetch address of control code table.

Action:

Get the address of the control code table.

Entry conditions:

No conditions.

Exit conditions:

HL contains the address of the control code table.
All other registers and flags preserved.

Notes:

All streams share one control code table so that any changes made to the table will affect all streams.

The control code table has a 3 byte entry for each control code. The entries are stored in ascending order, so the entry for #00 is first and that for #1F is last. The first byte of each entry is the number of parameters the control code requires, the other two bytes are the address of the routine to call to process the control code when all its parameters have been received. The routine must be located in the central 32K of RAM. It must obey the following interface:

Entry:

> A contains the last character added to the buffer.
> B contains the length of the buffer (including the control code).
> C contains the same as A.
> HL contains the address of the control code buffer (points at the control code).

Exit:

> AF, BC, DE, HL corrupt.
> All other registers preserved.

As the control code buffer only has space to store 9 parameter characters the number of parameters required should be limited to 9 or fewer.

The control code table is reinitialised to its default routines when TXT RESET is called.

Related Entries:

TXT OUTPUT

60: TXT STR SELECT #BBB4

Select a Text VDU stream.

Action:

Make a given stream the currently selected stream (if it isn't already).

Entry conditions:

A contains the required stream.

Exit conditions:

A contains the previously selected stream.

HL and flags corrupt.
All other registers preserved.

Notes:

The requested stream number is masked (with #07) to make it into a legal stream number.

Many attributes of the Text VDU may be set independently on different streams. It is important to ensure that the correct stream is selected when any of these are altered. These attributes are:

Pen ink.
Paper ink.
Cursor position.
Window limits.
Cursor enable/disable.
Cursor on/off.
VDU enable/disable.
Character write mode.
Graphic character write mode.

If the stream is already selected then this routine returns quickly. It is not unreasonable to repeatedly select a stream (before each character sent, for example).

Related Entries:

TXT OUTPUT

61: TXT SWAP STREAMS #BBB7

Swap the states of two streams.

Action:

The stream descriptors for two streams are exchanged. The currently selected stream number remains the same (although its descriptor may have been altered).

Entry conditions:

B contains a stream number.
C contains another stream number.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The stream numbers passed are masked (with #07) to ensure that they are legal stream numbers.

The attributes that are exchanged are:

- Pen ink.
- Paper ink.
- Cursor position.
- Window limits.
- Window roll count.
- Cursor enable/disable.
- Cursor on/off.
- VDU enable/disable.
- Character write mode.
- Graphic character write mode.

Related Entries:

TXT STR SELECT

62: GRA INITIALISE #BBBA

Initialise the Graphics VDU.

Action:

The Graphics VDU is fully initialised (as during EMS). All Graphic VDU variables and indirections are set to their default values.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The full operation is:

> Set the Graphics VDU indirections (GRA PLOT, GRA TEST and GRA LINE) to their default routines.
> Set the graphic paper to ink 0.
> Set the graphic pen to ink 1.
> Set the user origin to the bottom left corner of the screen.
> Move the current position to the user origin.
> Set the graphics window to cover the whole screen.
> The graphics window is not cleared.

Related entries:

GRA RESET
SCR INITIALISE

63: GRA RESET #BBBD

Reset the Graphics VDU.

Action:

Re-initialise the Graphics VDU indirections to their default routines.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Sets the Graphics VDU indirections (GRA PLOT, GRA TEST and GRA LINE) to their default routines. No other action is taken.

Related entries:

GRA INITIALISE

64: GRA MOVE ABSOLUTE #BBC0

Move to an absolute position.

Action:

Move the current position to an absolute position.

Entry conditions:

DE contains the required user X coordinate.
HL contains the required user Y coordinate.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The new position is given in user coordinates. i.e. Relative to the user origin.

The new position can be outside the graphics window.

The Graphic VDU plotting, testing and line drawing routines all move the current graphics position to the point (or endpoint) specified automatically.

Related entries:

GRA ASK CURSOR
GRA MOVE RELATIVE

65: GRA MOVE RELATIVE #BBC3

Move relative to current position.

Action:

Move the current position to relative to its current position.

Entry conditions:

DE contains a signed X offset.
HL contains a signed Y offset.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The new position can be outside the graphics window.

The Graphic VDU plotting, testing and line drawing routines all move the current graphics position to the point (or endpoint) specified automatically.

Related entries:

GRA ASK CURSOR
GRA MOVE ABSOLUTE

66: GRA ASK CURSOR #BBC6

Get the current position.

Action:

Ask where the current graphics position is.

Entry conditions:

No conditions.

Exit conditions:

DE contains the user X coordinate.
HL contains the user Y coordinate.

AF corrupt.
All other registers preserved.

Notes:

The current position is given in user coordinates. i.e. Relative to the user origin.

The Graphic VDU plotting, testing and line drawing routines all move the current graphics position to the point (or endpoint) specified automatically. Thus, the position returned is probably where the last point was plotted or tested.

Related entries:

GRA MOVE ABSOLUTE
GRA MOVE RELATIVE

67: GRA SET ORIGIN #BBC9

Set the origin of the user coordinates.

Action:

Set the location of the user origin and move the current position there.

Entry conditions:

DE contains the standard X coordinate of the origin.
HL contains the standard Y coordinate of the origin.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The origin position is given in standard coordinates in which (0,0) is the bottom left corner of the screen.

The default origin position is at (0,0). Whenever the screen mode is changed, by calling SCR SET MODE, the origin is restored to its default position.

Related entries:

GRA GET ORIGIN

68: GRA GET ORIGIN #BBCC

Get the origin of the user coordinates.

Action:

Ask where the user coordinate origin is located.

Entry conditions:

No conditions.

Exit conditions:

DE contains the standard X coordinate of the origin.
HL contains the standard Y coordinate of the origin.

All other registers preserved.

Notes:

The origin position is given in standard coordinates in which (0,0) is the bottom left corner of the screen.

Related entries:

GRA SET ORIGIN

69: GRA WIN WIDTH #BBCF

Set the right and left edges of the graphics window.

Action:

Set the horizontal position of the graphics window. The left and right edges are respectively the first and last points that lie inside the window horizontally.

Entry conditions:

DE contains the standard X coordinate of one edge.
HL contains the standard X coordinate of the other edge.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The window edges are given in standard coordinates - in which (0,0) is the bottom left corner of the screen and coordinates are signed 16 bit numbers.

The left edge of the window is deemed to be the smaller of the two edges supplied.

The window will be truncated, if necessary, to make it fit the screen. The edges are moved to screen byte boundaries so that the window only contains whole bytes (the left edge is moved left, the right edge is moved right). This moves the coordinates of the edges as follows in the various modes:

Mode	Left Edge	Right Edge
0	Multiple of 2	Multiple of 2 minus 1
1	Multiple of 4	Multiple of 4 minus 1
2	Multiple of 8	Multiple of 8 minus 1

The default window covers the whole screen. Whenever the screen mode is changed the window is restored to its default size.

All Graphics VDU point plotting and line drawing routines test whether the points they are about to plot lie inside the window; if they are not then the points are not plotted.

Related entries:

GRA GET W WIDTH
GRA WIN HEIGHT

70: GRA WIN HEIGHT #BBD2

Set the top and bottom edges of the graphics window.

Action:

Set the vertical position of the graphics window. The top and bottom edges are respectively the last and first points that lie inside the window vertically.

Entry conditions:

DE contains the standard Y coordinate of one edge.
HL contains the standard Y coordinate of the other edge.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The window edges are given in standard coordinates - in which (0,0) is the bottom left corner of the screen and coordinates are signed 16 bit numbers.

The top edge will be deemed to be the higher of the two edges supplied.

The window will be truncated, if necessary, to make it fit the screen. The edges will be moved to lie on screen line boundaries so that only whole screen lines are included in the window (the top edge will be moved up, the bottom edge will be moved down). This moves the bottom edge to an even coordinate and the top edge to an odd coordinate.

The default window covers the whole screen. Whenever the screen mode is changed the window is restored to its default size.

All Graphics VDU point plotting and line drawing routines test whether the points they are about to plot lie inside the window; if they do not then the points are not plotted.

Related entries:

GRA GET W HEIGHT
GRA WIN WIDTH

71: GRA GET W WIDTH #BBD5

Get the left and right edges of the graphics window.

Action:

Ask the horizontal position of the graphics window. The left and right edges are respectively the first and last points that lie inside the window horizontally.

Entry conditions:

No conditions.

Exit conditions:

DE contains the standard X coordinate of the left edge of the window.
HL contains the standard X coordinate of the right edge of the window.

AF corrupt.
All other registers preserved.

Notes:

The window edges are given in standard coordinates in which (0,0) is the bottom left corner of the screen.

The edges may not be exactly the same as those that were set using GRA WIN WIDTH as the window is truncated to fit the screen and the edges are moved to screen byte boundaries so that the window only contains whole bytes .

Related entries

GRA GET W HEIGHT
GRA WIN WIDTH

72: GRA GET W HEIGHT #BBD8

Get the top and bottom edges of the graphics window.

Action:

Ask the vertical position of the graphics window. The top and bottom edges are respectively the last and first points that lie inside the window vertically.

Entry conditions:

No conditions.

Exit conditions:

DE contains the standard Y coordinate of the top edge of the window.
HL contains the standard Y coordinate of the bottom edge of the window.

AF corrupt.
All other registers preserved.

Notes:

The window edges are given in standard coordinates. i.e. With (0,0) being the bottom left corner of the screen.

The edges may not be exactly the same as those passed to GRA WIN HEIGHT as the window is truncated to fit the screen and the edges are moved to lie on screen line boundaries so that only whole screen lines are included in the window.

Related entries:

GRA GET W WIDTH
GRA WIN HEIGHT

73: GRA CLEAR WINDOW #BBDB

Clear the graphic window.

Action:

Clear the graphics window to the graphics paper ink.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The current graphics position is moved to the origin of the user coordinates.

Related entries:

GRA SET PAPER
GRA WIN HEIGHT
GRA WIN WIDTH
SCR CLEAR
TXT CLEAR WINDOW

74: GRA SET PEN #BBDE

Set the graphics plotting ink.

Action:

Set the graphics pen ink. This is the ink used by the Graphics VDU for plotting points, drawing lines and writing characters.

Entry conditions:

A contains the required ink.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The ink is masked to bring it in the range of inks for the current screen mode. In mode 0 the mask is #0F, in mode 1 it is #03 and in mode 2 it is #01.

Related entries:

GRA GET PEN
GRA SET PAPER
SCR SET INK
TXT SET PEN

75: GRA GET PEN #BBE1

Get the current graphics plotting ink.

Action:

Ask what the current graphics pen ink is set to. This is the ink used by the Graphics VDU for plotting points, drawing lines and writing characters.

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

GRA GET PAPER
GRA SET PEN
SCR GET INK
TXT GET PEN

76: GRA SET PAPER #BBE4

Set the graphics background ink.

Action:

Set the graphics paper ink.

Entry conditions:

A contains the required ink.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The ink is masked to bring it in the range of inks for the current screen mode. In mode 0 the mask is #0F, in mode 1 it is #03 and in mode 2 it is #01.

The paper ink is the ink that is used for clearing the graphics window and writing the background to characters. It is assumed to cover everywhere outside the graphics window when testing points.

Related entries:

GRA GET PAPER
GRA SET PEN
SCR GET INK
TXT SET PAPER

77: GRA GET PAPER #BBE7

Get the current graphics background ink.

Action:

Ask what the current graphics paper ink is set to.

Entry conditions:

No conditions.

Exit conditions:

A contains the ink.

Flags corrupt.
All other registers preserved.

Notes:

The paper ink is the ink that is used for clearing the graphics window and writing the background to characters. It is assumed to cover everywhere outside the graphics window when testing points.

Related entries:

GRA GET PEN
GRA SET PAPER
SCR GET INK
TXT GET PAPER

78: GRA PLOT ABSOLUTE #BBEA

Plot a point at an absolute position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the point is plotted in the current graphics pen ink using the current graphics write mode. If the point lies outside the graphics window then no action is taken.

Entry conditions:

DE contains the user X coordinate to plot at.
HL contains the user Y coordinate to plot at.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The position to plot at is given in user coordinates. i.e. Relative to the user origin.

This routine calls the GRA PLOT indirection to plot the point. In its turn GRA PLOT calls the SCR WRITE indirection to set the pixel (if it is in the window).

Related entries:

GRA PLOT
GRA PLOT RELATIVE
GRA TEST ABSOLUTE

79: GRA PLOT RELATIVE #BBED

Plot a point relative to the current position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the point is plotted in the current graphics pen ink using the current graphics write mode. If the point lies outside the graphics window then no action is taken.

Entry conditions:

DE contains a signed X offset.
HL contains a signed Y offset.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The position to plot at is given in relative coordinates. i.e. Relative to the current graphics position.

This routine calls the GRA PLOT indirection to plot the point. In its turn GRA PLOT calls the SCR WRITE indirection to set the pixel (if it is in the window).

Related entries:

GRA PLOT
GRA PLOT ABSOLUTE
GRA TEST RELATIVE

80: GRA TEST ABSOLUTE #BBF0

Test a point at an absolute position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the pixel is read from the screen and the ink it is set to is decoded and returned. If the position lies outside the graphics window then the current paper ink is returned.

Entry conditions:

DE contains the user X coordinate to test at.
HL contains the user Y coordinate to test at.

Exit conditions:

A contains the ink of the specified point (or the graphics paper ink).

BC, DE, HL and flags corrupt.
All other registers preserved.

Notes:

The position to test is given in user coordinates. i.e. Relative to the user origin.

This routine calls the GRA TEST indirection to test the point. In its turn GRA TEST calls the SCR READ indirection to test the pixel (if it is in the window).

Related entries:

GRA PLOT ABSOLUTE
GRA TEST
GRA TEST RELATIVE

81: GRA TEST RELATIVE #BBF3

Test a point relative to the current position.

Action:

The current graphic position is moved to the position supplied. If this lies inside the graphics window then the pixel is read from the screen and the ink it is set to is decoded and returned. If the position lies outside the graphics window then the current paper ink is returned.

Entry conditions:

DE contains a signed X offset.
HL contains a signed Y offset.

Exit conditions:

A contains the ink of the specified point (or the graphics paper ink).

BC, DE, HL and flags corrupt.
All other registers preserved.

Notes:

The position to test is given in relative coordinates. i.e. Relative to the current graphics position.

This routine calls the GRA TEST indirection to test the point. In its turn GRA TEST calls the SCR READ indirection to test the pixel (if it is in the window).

Related entries:

GRA PLOT RELATIVE
GRA TEST
GRA TEST ABSOLUTE

82: GRA LINE ABSOLUTE #BBF6

Draw a line to an absolute position.

Action:

Move the current graphics position to the endpoint supplied. All points between this position and the previous graphics position that lie inside the graphics window will be plotted in the current graphics pen ink using the current graphics write mode. Points that lie outside the graphics window are ignored.

Entry conditions:

DE contains the user X coordinate of the endpoint.
HL contains the user Y coordinate of the endpoint.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The position of the end of the line is given in user coordinates. i.e. Relative to the user origin.

This routine calls the GRA LINE indirection to draw the line. In its turn GRA LINE calls the SCR WRITE indirection to write the pixels (for pixels in the graphics window).

Related entries:

GRA LINE
GRA LINE RELATIVE

83: GRA LINE RELATIVE #BBF9

Draw a line relative to the current position.

Action:

Move the current graphics position to the endpoint supplied. All points between this position and the previous graphics position that lie inside the graphics window will be plotted in the current graphics pen ink using the current graphics write mode. Points that lie outside the graphics window are ignored.

Entry conditions:

DE contains the signed X offset of the endpoint.
HL contains the signed Y offset of the endpoint.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The position of the end of the line is given in relative coordinates, i.e. Relative to the current graphics position.

This routine calls the GRA LINE indirection to draw the line. In its turn GRA LINE calls the SCR WRITE indirection to write the pixels (for pixels in the graphics window).

Related entries:

GRA LINE
GRA LINE ABSOLUTE

84: GRA WR CHAR #BBFC

Put a character on the screen at the current graphics position.

Action:

Write a character on the screen at the current graphics position.

Entry conditions:

A contains the character to write.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The character is written with its top left corner being the current graphics position.

All characters are printed, even control codes (characters #00..#1F).

The current position is moved right by the width of the character (ready for another character to be written). In mode 0 this move is 32 points right, in mode 1 the move is 16 points and in mode 0 it is 8 points.

The character will be written in the current graphic pen ink and the background will be written in the current graphic paper ink. The background will always be written even if the Text VDU is writing characters in transparent mode. Pixels in the character that lie outside the graphics window will not be plotted. The pixels are plotted using the SCR WRITE indirection so they are written using the current graphics write mode.

Related entries:

TXT SET GRAPHIC
TXT WR CHAR

85: SCR INITIALISE #BBFF

Initialise the Screen Pack.

Action:

Full initialisation of the Screen Pack (as used during EMS). All Screen Pack variables and indirections are initialised, also the screen mode and the inks are initialised to their default settings.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The screen indirections (SCR READ, SCR WRITE and SCR MODE CLEAR) are set to their default routines .
The inks are set to their default colours (see Appendix V).
The ink flashing periods are set to their default values.
The screen is put into mode 1.
The screen base is set to put the screen memory at #C000..#FFFF (under the upper ROM).
The screen offset is set to 0.
The screen is cleared to ink 0.
The Graphics VDU write mode is set to FORCE mode.
The ink flashing frame flyback event is set up.

The initialisation is performed in an order that attempts to avoid the previous contents of the screen becoming visible (at EMS the contents will be random).

Related entries:

GRA INITIALISE
SCR RESET
TXT INITIALISE

86: SCR RESET #BC02

Reset the Screen Pack.

Action:

Re-initialises the Screen Pack indirections and the ink colours. Also re-initialises the flash rate and Graphics VDU write mode.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The screen indirections (SCR READ, SCR WRITE and SCR MODE CLEAR) are set to their default routines.
The inks are set to their default colours (see Appendix V).
The ink flashing periods are set to their default values.
The Graphics VDU write mode is set to FORCE mode.

The inks are not passed to the hardware. This will be done when the inks flash next.

Related entries:

SCR INITIALISE
SCR SET ACCESS
SCR SET FLASHING
SCR SET INK

87: SCR SET OFFSET #BC05

Set the offset of the start of the screen.

Action:

Set the offset of the first character on the screen. By changing this offset the screen can be rolled.

Entry conditions:

HL contains the required offset.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The offset passed is masked with #07FE to make sure it is not too big and to make sure that the offset is even. (The screen is only capable of rolling in two byte increments).

The screen base and screen offset are combined into a single value and sent to the hardware together.

The screen offset is used by SCR CHAR POSITION and SCR DOT POSITION to calculate screen addresses. If the screen offset is changed merely by calling the Machine Pack routine MC SCREEN OFFSET then the Text and Graphics VDUs will use incorrect screen addresses.

The offset is set to zero when the screen mode is set or the screen is cleared by calling SCR CLEAR.

Related entries:

MC SCREEN OFFSET
SCR GET LOCATION
SCR HW ROLL
SCR SET BASE

88: SCR SET BASE #BC08

Set the area of RAM to use for the screen memory.

Action:

Sets the base address of the screen memory. This can be used to move the screen out from underneath the upper ROM or to display a prepared screen instantly.

Entry conditions:

A contains the more significant byte of the base address.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The screen memory can only be located on a 16K boundary so the value passed is masked with #C0. The default screen base, set at EMS, is #C0.

The screen offset is combined with the screen base into a single value which is sent to the hardware.

The screen base address is used by SCR CHAR POSITION and SCR DOT POSITION to calculate screen addresses. If the screen base is changed merely by calling the Machine pack routine MC SCREEN OFFSET then the text and graphics VDUs will use incorrect screen addresses.

The screen memory is not cleared when the screen base is set, use SCR CLEAR to do this.

Related entries:

MC SCREEN OFFSET
SCR GET LOCATION
SCR SET OFFSET

89: SCR GET LOCATION #BC0B

Fetch current base and offset settings.

Action:

Ask where the screen memory is located and where the start of the screen is.

Entry conditions:

No conditions.

Exit conditions:

A contains the more significant byte of the base address.
HL contains the current offset.

Flags corrupt.
All other registers preserved.

Notes:

The base and offsets returned by this routine may not be the same as those set using SCR SET BASE or SCR SET OFFSET. This is because the values are masked to make them legal and the screen offset is also changed when the hardware screen rolling routine, SCR HW ROLL, is used.

Related entries:

SCR SET BASE
SCR SET OFFSET

90: SCR SET MODE #BC0E

Set screen into a new mode.

Action:

Put the screen into a new mode and make sure that the Text and Graphics VDUs are set up correctly.

Entry conditions:

A contains the required mode.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The mode requested is masked with #03. If the resulting value is 3 then no action is taken. Otherwise one of the following screen modes is set up:

Mode 0:	160 x 200 pixels,	20 x 25 characters.
Mode 1:	320 x 200 pixels,	40 x 25 characters.
Mode 2:	640 x 200 pixels,	80 x 25 characters.

At an early stage the screen is cleared to avoid the old contents of the screen being displayed in the wrong mode. The screen is cleared by calling the SCR MODE CLEAR indirection.

All text and graphics windows are set to cover to whole screen and the graphics user origin is set to the bottom left corner of the screen. The cursor blobs for all text streams are turned off.

The current text and graphics pen and paper inks are masked as appropriate for the new mode (see TXT SET PEN et al). When changing mode to a mode that allows fewer inks on the screen this may cause the pen or paper inks to change.

Related entries:

MC SET MODE
SCR GET MODE

91: SCR GET MODE   #BC11

Ask the current screen mode.

Action:

Fetch and test the current screen mode.

Entry conditions:

No conditions.

Exit conditions:

If current mode is mode 0:

Carry true.
Zero false.
A contains 0.

If current mode is mode 1:

Carry false.
Zero true.
A contains 1.

If current mode is mode 2:

Carry false.
Zero false.
A contains 2.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

The modes are:

Mode 0:	160 x 200 pixels,	20 x 25 characters.
Mode 1:	320 x 200 pixels,	40 x 25 characters.
Mode 2:	640 x 200 pixels,	80 x 25 characters.

Related entries:

SCR SET MODE

92: SCR CLEAR #BC14

Clear the screen (to ink zero).

Action:

Clear the whole of screen memory to zero.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

At an early stage the ink flashing is turned off and the inks are all set to the same colour as ink 0. This makes the screen clearing appear instantaneous. When all the screen memory has been set to 0 the ink flashing is turned back on (an ink flashing event is added to the frame flyback queue) and all inks are set to their proper colours.

If the text paper ink and the graphics paper ink are not set to ink 0 then this will become apparent on the screen when characters are written or windows are cleared.

The screen offset is set to zero.

Related entries:

GRA CLEAR WINDOW
SCR MODE CLEAR
TXT CLEAR WINDOW

93: SCR CHAR LIMITS #BC17

Ask the size of the screen in characters.

Action:

Get the last character row and column on the screen in the current mode.

Entry conditions:

No conditions.

Exit conditions:

B contains the physical last column on the screen.
C contains the physical last row on the screen.

AF corrupt.
All other registers preserved.

Notes:

The screen edges are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. This means that the last column on the screen is 19 in mode 0, 39 in mode 1 and 79 in mode 2. The last row on the screen is 24 in all modes.

Related entries:

SCR GET MODE

94: SCR CHAR POSITION #BC1A

Convert physical coordinates to a screen position.

Action:

Calculate the screen address of the top left corner of a character position on the screen. Also return the width of a character in the current mode.

Entry conditions:

H contains the physical character column.
L contains the physical character row.

Exit conditions:

HL contains the screen address of the top left corner of the character.
B contains the width in bytes of a character in screen memory.

AF corrupt.
All other registers preserved.

Notes:

The character position is given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen.

The character position given is not checked for being legal. An illegal position (one outside the limits of the screen) will generate a meaningless screen address.

The conversion to screen address uses the following formula:

Screen address = Screen base + (Block offset MOD #0800)

where:

Block offset = (Row * 80) + (Column * Width) + Screen offset

and:

Screen base	is the address of the start of screen memory.
Width	is the width of a character in bytes in the current mode (4 in mode 0, 2 in mode 1, 1 in mode 2).
Screen offset	is offset of the first byte to be displayed on the screen.

Related entries:

SCR DOT POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV BYTE
SCR PREV LINE

95: SCR DOT POSITION #BC1D

Convert base coordinates to a screen position.

Action:

Calculate the screen address and mask for a pixel. Also return an indication of the number of pixels in a screen byte in the current mode.

Entry conditions:

DE contains the base X coordinate of a pixel.
HL contains the base Y coordinate of a pixel.

Exit conditions:

HL contains the screen address of the pixel.
C contains the mask for the pixel.
B contains one less than the number of pixels in a byte.

AF and DE corrupt.
All other registers preserved.

Notes:

The pixel position is given in base coordinates. i.e. (0,0) is the pixel in the bottom left corner of the screen and each coordinate position refers to a single pixel.

The pixel position is not checked for being legal (within the limits of the screen). If it is not then the screen address calculated is meaningless.

The conversion to screen address uses the following formula:

Screen address = Screen base + (Line in row * #0800) + (Row offset MOD #0800)

where:

Screen base	is the start address of screen memory.
Line in row	= (199 – Y coordinate) MOD 8
Row offset	= (Row number * 80) + Byte in row + Screen offset

and:

Row number	= (199 – Y coordinate) / 8
Byte in row	= X coordinate / Byte width
Screen offset	is offset of the first byte to be displayed on the screen.

Byte width	is the number of pixels in a byte in the current mode (2 in mode 0, 4 in mode 1, 8 in mode 2).

X coordinate MOD Byte width is used to calculate the mask for the appropriate pixel.

Related entries:

SCR CHAR POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV BYTE
SCR PREV LINE

96: SCR NEXT BYTE #BC20

Step a screen address right one byte.

Action:

Calculate the screen address of the byte right of the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the end of the screen line is not prevented. It will simply point the screen address at the next byte in the screen block. Normally this will be the first byte on a screen line 8 screen lines down from the old line (i.e. down one character row). However, moving right off the end of the last screen line in a block will point the screen address at the start of the 48 bytes in the block that are not displayed on the screen.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT LINE
SCR PREV BYTE
SCR PREV LINE

97: SCR PREV BYTE #BC23

Step a screen address left one byte.

Action:

Calculate the screen address of the byte left of the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the start of the screen line is not prevented. It will simply point the screen address at the previous byte in the screen block. Normally this will be the last byte on a screen line 8 screen lines up from the old line (i.e. up one character row). However, moving left off the start of the top screen line in a block will point the screen address at the last of the 48 bytes in the block that are not displayed on the screen.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV LINE

98: SCR NEXT LINE #BC26

Step a screen address down one line.

Action:

Calculate the screen address of the byte below the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the bottom of the screen is not prevented (and not recommended). After moving off the bottom the screen address is not useful.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT BYTE
SCR PREV BYTE
SCR PREV LINE

99: SCR PREV LINE #BC29

Step a screen address up one line.

Action:

Calculate the screen address of the byte above the supplied screen address.

Entry conditions:

HL contains a screen address.

Exit conditions:

HL contains the updated screen address.

AF corrupt.
All other registers preserved.

Notes:

Moving off the top of the screen is not prevented (and not recommended). After moving off the top the screen address is not useful.

This routine is intended to be used for moving the screen address when putting characters or drawing lines on the screen.

Related entries:

SCR CHAR POSITION
SCR DOT POSITION
SCR NEXT BYTE
SCR NEXT LINE
SCR PREV BYTE

100: SCR INK ENCODE #BC2C

Encode an ink to cover all pixels in a byte.

Action:

Convert an ink to the encoded form that will set all pixels in a byte to the ink. This encoded ink can then be masked to generate the appropriate value to set a single pixel to the ink.

Entry conditions:

A contains an ink number.

Exit conditions:

A contains the encoded ink.

Flags corrupt.
All other registers preserved.

Notes:

The encoding is not trivial as the pixels in a byte are interleaved and also the bits in a pixel are not in the obvious order. The pixel bits are (most significant to least significant):

	Mode 0	Mode 1	Mode 2
Leftmost pixel:	Bits 1,5,3,7	Bits 3,7	Bit 7
			Bit 6
		Bits 2,6	Bit 5
			Bit 4
	Bits 0,4,2,6	Bits 1,5	Bit 3
			Bit 2
		Bits 0,4	Bit 1
Rightmost pixel:			Bit 0

The Text and Graphic VDUs store their pen and paper inks in this encoded form for ease of use internally. This saves time converting the ink for each pixel plotted.

The encoding is different in different modes and so all inks have to be re-encoded when the screen mode is changed. SCR SET MODE does this automatically for the Text VDU and Graphics VDU pen and paper inks.

Related entries:

SCR INK DECODE

101: SCR INK DECODE #BC2F

Decode an encoded ink.

Action:

Convert an encoded ink to the appropriate ink number.

Entry conditions:

A contains an encoded ink.

Exit conditions:

A contains the ink number.

Flags corrupt.
All other registers preserved.

Notes:

The decoding is performed by decoding the ink of the leftmost pixel in the encoded ink. The ink for this pixel is encoded in the following bits (most significant to least significant) in the various screen modes:

Mode 0:	Bits 1,5,3,7
Mode 1:	Bits 3,7
Mode 2:	Bit 7

Related entries:

SCR INK ENCODE

102: SCR SET INK #BC32

Set the colours in which to display an ink.

Action:

Set which two colours will be used to display an ink. If the two colours are the same then the ink will remain a steady colour. If the colours are different then the ink will alternate between these two colours.

Entry conditions:

A contains an ink number.
B contains the first colour.
C contains the second colour.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The ink number is masked with #0F to make sure it is legal, and the colours are masked with #1F. Colours 27..31 are not intended for use; they are merely duplicates of other colours available.

The new colours for an ink are not sent to the hardware immediately. They are stored and will appear on the screen when the next frame flyback occurs.

The length of time for which each colour is displayed on the screen can be set by calling SCR SET FLASHING.

The inks are set to their default colours at EMS and when SCR RESET is called.

The various colours available and the default ink colours set are described in Appendix V.

Related entries:

GRA SET PAPER
GRA SET PEN
SCR GET INK
SCR SET BORDER
SCR SET FLASHING
TXT SET PAPER
TXT SET PEN

103: SCR GET INK #BC35

Ask the colours an ink is currently displayed in.

Action:

Get the two colours that are used to display an ink on the screen.

Entry conditions:

A contains an ink number.

Exit conditions:

B contains the first colour.
C contains the second colour.

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The ink number is masked with #0F to make sure it is legal. The colours returned may not be the same as those supplied to the Screen Pack as the colours are masked when they are set.

The new colours for an ink are not sent to the hardware immediately when they are set. They are stored and appear on the screen when the next frame flyback occurs. This means that the colours returned may not actually be visible to the user yet.

The default settings for the inks and the various colours available are described in Appendix V.

Related entries:

GRA GET PAPER
GRA GET PEN
SCR GET BORDER
SCR SET INK
TXT GET PAPER
TXT GET PEN

104: SCR SET BORDER #BC38

Set the colours in which to display the border.

Action:

Set which two colours will be used to display the border. If the two colours are the same then the border will remain a steady colour. If the colours are different then the border will alternate between these two colours.

Entry conditions:

B contains the first colour.
C contains the second colour.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The colours are masked with #1F to ensure that they are legal. Colours 27..31 are not intended for use; they are merely duplicates of other colours available.

The new colours for the border are not sent to the hardware immediately. They are stored and will appear on the screen when the next frame flyback occurs.

The length of time for which each colour is displayed on the screen can be set by calling SCR SET FLASHING.

The border is set to its default colour at EMS and when SCR RESET is called. The default colour and the colours available are described in Appendix V.

Related entries:

SCR GET BORDER
SCR SET FLASHING
SCR SET INK

106: SCR SET FLASHING #BC3E

Set the flash periods.

Action:

Set for how long each of the two colours for the inks and the border are to be displayed on the screen. These settings apply to all inks and the border.

Entry conditions:

H contains the period for the first colour.
L contains the period for the second colour.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The flash periods are given in frame flybacks (1/50 or 1/60 of a second). A period of 0 is taken to mean a period of 256.

The default setting for the flash periods is 10 frame flybacks (1/5 or 1/6 of a second). This is set at EMS and when SCR RESET is called.

The new flash periods are not used immediately but when the inks next flash.

Related entries:

SCR GET FLASHING
SCR SET BORDER
SCR SET INK

107: SCR GET FLASHING #BC41

Ask the current flash periods.

Action:

Get the time for which each of the two colours associated with an ink or the border is displayed.

Entry conditions:

No conditions.

Exit conditions:

H contains the period for the first colour.
L contains the period for the second colour.

AF corrupt.
All other registers preserved.

Notes:

The flash periods are given in frame flybacks (1/50 or 1/60 of a second).

A period of 0 means 256.

Related entries:

SCR SET FLASHING

108: SCR FILL BOX #BC44

Fill a character area of the screen with an ink.

Action:

Fill a rectangular area of the screen with an ink. The boundaries of this area are given in character positions.

Entry conditions:

A contains the encoded ink to fill the area with.
H contains the physical left column of the area to fill.
D contains the physical right column of the area to fill.
L contains the physical top row of the area to fill.
E contains the physical bottom row of the area to fill.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The area boundaries are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. They are not checked for legality. If illegal boundaries are passed (edges off the screen) then unpredictable effects may occur.

The screen is written directly without using any other write routine. The current Graphics VDU write mode is therefore ignored.

Related entries:

SCR CLEAR
SCR FLOOD BOX
TXT CLEAR WINDOW

109: SCR FLOOD BOX #BC47

Fill a byte area of the screen.

Action:

Fill a rectangular area of the screen with an ink. The boundaries of the area must lie on byte boundaries. This routine will not fill an arbitrary area of the screen to a pixel boundary.

Entry conditions:

C contains the encoded ink to fill the area with.
HL contains the screen address of the top left corner of the area to fill.
D contains the (unsigned) width of the area to fill in bytes.
E contains the (unsigned) height of the area to fill in screen lines.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The whole of the rectangle being cleared must lie on the screen. If any of it lies off the screen then unpredictable effects may occur.

A height or width of 0 is taken to mean 256 (which is too large to fit on the screen).

The screen is written directly without using any other write routine. The current Graphics VDU write mode is therefore ignored.

Related entries:

GRA CLEAR WINDOW
SCR CLEAR
SCR FILL BOX

110: SCR CHAR INVERT #BC4A

Invert a character position.

Action:

All pixels at a character position that are written in one ink are rewritten in a second ink, and vice versa. This gives an inverse effect to the character position. Inverting the character a second time will restore the original inks. This effect is used to draw the Text VDU cursors.

Entry conditions:

B contains an encoded ink.
C contains another encoded ink.
H contains a physical character column.
L contains a physical character row.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The character position is given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen.

The character position given is not checked for being legal. An illegal position (one outside the limits of the screen) will have unpredictable effects.

All pixels at the character position are exclusive-ored with the exclusive-or of the two inks supplied. Pixels at the character position that are set to one of the two inks supplied will therefore be set to the other supplied ink. Pixels set to other inks will also be altered.

Related entries:

TXT PLACE CURSOR
TXT REMOVE CURSOR

111: SCR HW ROLL #BC4D

Move the whole screen up or down eight pixel lines (one character).

Action:

Roll the screen using the hardware. The new line appearing on the screen is cleared.

Entry conditions:

If the screen is to roll down:

B must be zero.

If the screen is to roll up:

B must be non-zero.

Always:

A contains the encoded ink to clear the new line to.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The screen is rolled by changing the screen offset (see SCR SET OFFSET).

Rolling the screen upwards moves the screen contents up and clears the new bottom line. The screen offset is therefore increased by 80 (MOD #0800).

Rolling the screen downwards moves the screen contents down and clears the new top line. The screen offset is therefore decreased by 80 (MOD #0800).

The new line is cleared by writing to it directly thus the Graphics VDU write mode is ignored.

The Text VDU roll count is not changed by this routine (see TXT GET WINDOW).

Special precautions are taken to make sure that the screen is kept looking presentable during the rolling and in particular during the clearing of the new line. Principally this consists of clearing the new line in two parts. First the part that is not visible on the screen (by virtue of the screen addressing) is cleared. Then, after waiting for frame flyback and changing the screen offset, the second half of the the line that was part of the line that just rolled off the screen is cleared.

Related entries:

SCR SET OFFSET
SCR SW ROLL

112: SCR SW ROLL #BC50

Move an area of the screen up or down eight pixel lines (one character).

Action:

Roll an area of the screen by copying. The area to be rolled is specified in character positions.

Entry conditions:

If the screen is to roll down:

B must be zero.

If the screen is to roll up:

B must be non-zero.

Always:

A contains the encoded ink to clear the new line to.
H contains the physical left column of the area to roll.
D contains the physical right column of the area to roll.
L contains the physical top row of the area to roll.
E contains the physical bottom row of the area to roll.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The area boundaries are given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. The boundaries are not checked for legality. If illegal boundaries are passed (edges off the screen) then unpredictable effects may occur.

Rolling the area upwards moves the area contents up and clears the new bottom line. Rolling the area downwards moves the area contents down and clears the new top line.

The line is cleared by writing to it directly; the Graphics VDU write mode is ignored.

The Text VDU roll count is not changed by this routine (see TXT GET WINDOW).

Special precautions are taken to make sure that the screen is kept looking presentable during the rolling. Principally this consists of waiting for frame flyback before performing the copy.

Related entries:

SCR HW ROLL

113: SCR UNPACK #BC53

Expand a character matrix for the current screen mode.

Action:

Convert a matrix from its standard form to a set of pixel masks as appropriate for the current screen mode.

Entry conditions:

HL contains the address of a matrix.
DE contains the address of an area to unpack into.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The matrix is converted into a series of masks which cover all the screen bytes in the character. This means that each byte of the matrix is converted to 4 bytes in mode 0, 2 bytes in mode 1 and 1 byte in mode 2. Thus the unpacking area must be 32, 16 or 8 bytes long.

If a bit in the matrix is set then the appropriate pixel mask is included in the unpacked version (the bits are set to one). Otherwise the pixel mask is not included in the unpacked version (the bits are set to zero).

Related entries:

SCR REPACK

114: SCR REPACK #BC56

Compress a character matrix to the standard form.

Action:

A character on the screen is converted to a matrix by comparing each pixel with an ink. If the pixel is set to that ink then the appropriate bit in the character matrix is set, otherwise the bit is cleared.

Entry conditions:

A contains the encoded ink to match against.
H contains the physical character column to read from.
L contains the physical character row to read from.
DE contains the address of the area to construct the matrix in.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The character position is given in physical coordinates in which row 0, column 0 is the top left corner of the screen.

The character position given is not checked for legality. An illegal position (one outside the limits of the screen) will have unpredictable effects.

The matrix produced has the normal layout. It is 8 bytes long, stored top line first and bottom line last, the most significant bit of a byte refers to the leftmost pixel of a line and the least significant bit to the rightmost pixel.

Because the pixels are tested for being set to only one ink the matrix produced is not an exact representation of what is on the screen. It may be necessary, when trying to read characters from the screen, to repack using various different inks.

Related entries:

SCR UNPACK
TXT RD CHAR

115: SCR ACCESS #BC59

Set the screen write mode for the Graphics VDU.

Action:

Set the Graphics VDU write mode so that the Graphics VDU plots pixels by writing, anding, oring or exclusive-oring.

Entry conditions:

A contains the required write mode.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The write mode is masked with #03 to make it legal. The write modes are:

0: FORCE mode:	NEW = INK
1: XOR mode:	NEW = INK exclusive-or OLD
2: AND mode:	NEW = INK and OLD
3: OR mode:	NEW = INK or OLD

NEW is the final setting of the pixel.
OLD is the current setting of the pixel.
INK is the ink being plotted.

The default mode is FORCE mode (mode 0) and this is set at EMS and when SCR RESET is called.

Setting the write mode affects how the indirection routine SCR WRITE sets pixels. Graphics VDU plotting routines call this indirection to set pixels and so the write mode affects the Graphics VDU. No Text VDU routines call this indirection (they set pixels on the screen directly) and so the write mode does not affect the Text VDU. The routines that clear areas of the screen (e.g. GRA CLEAR WINDOW) act like the Text VDU and are unaffected by the write mode.

Related entries:

SCR WRITE

116: SCR PIXELS #BC5C

Write a pixel to the screen ignoring the Graphics VDU write mode.

Action:

Write a pixel or pixels to the screen. The position to write at is given by a screen address and pixel mask. The pixel is always set to the ink supplied whatever mode of writing the Graphics VDU is using.

Entry conditions:

B contains the encoded ink to write.
C contains the mask for the pixel(s).
HL contains the screen address of the pixel(s).

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The screen address is not checked and so passing an invalid screen address will have unpredictable results.

The pixel mask may be a combined mask for more than one pixel (thus speeding up plotting in certain cases).

To plot a pixel using the Graphics VDU write mode SCR WRITE should be called. SCR PIXELS is equivalent to calling SCR WRITE when the default mode (FORCE mode) is selected. The Text VDU sets the pixels in characters using FORCE mode.

Related entries:

SCR WRITE

117: SCR HORIZONTAL #BC5F

Plot a purely horizontal line.

Action:

Draw a line on the screen that runs horizontally. The pixels on the line are plotted using the SCR WRITE indirection and thus use the current Graphics VDU write mode.

Entry conditions:

A contains the encoded ink to draw in.
DE contains the base X coordinate of the start of the line.
BC contains the base X coordinate of the end of the line.
HL contains the base Y coordinate of the line.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The endpoints of the line are given in base coordinates, i.e. (0,0) is the pixel in the bottom left corner of the screen and each coordinate position refers to a single pixel.

The endpoints are not checked for being legal (within the limits of the screen). If they are not legal then unpredictable effects may occur.

The start X coordinate must be less than or equal to the end X coordinate.

This routine may be used to duplicate the method that the Graphics VDU uses for plotting lines - it splits a line that is more horizontal than vertical into a number of segments that are purely horizontal and plots these separately.

Related entries:

GRA LINE ABSOLUTE
GRA LINE RELATIVE
SCR VERTICAL

118: SCR VERTICAL #BC62

Plot a purely vertical line.

Action:

Draw a line on the screen that runs vertically. The SCR WRITE indirection is used to plot pixels in the line thus the current Graphics VDU write mode is used.

Entry conditions:

A contains the encoded ink to draw in.
DE contains the base X coordinate of the line.
HL contains the base Y coordinate of the start of the line.
BC contains the base Y coordinate of the end of the line.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The endpoints of the line are given in base coordinates. i.e. (0,0) is the pixel in the bottom left corner of the screen and each coordinate position refers to a single pixel.

The endpoints are not checked for being legal (within the limits of the screen). If they are not legal then unpredictable effects may occur.

The start Y coordinate must be less than or equal to the end Y coordinate.

This routine may be used to duplicate the method that the Graphics VDU uses for plotting lines - it splits a line that is more vertical than horizontal into a number of segments that are purely vertical and plots these separately.

Related entries:

GRA LINE ABSOLUTE
GRA LINE RELATIVE
SCR HORIZONTAL

119: CAS INITIALISE #BC65

Initialise the Cassette Manager.

Action:

Full initialisation of the Cassette Manager (as used during EMS).

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Operations carried out are:

All streams are marked closed.
The default write speed is set up.
The prompt messages are turned on.

Related entries:

CAS IN ABANDON
CAS NOISY
CAS OUT ABANDON
CAS SET SPEED

120: CAS SET SPEED #BC68

Set the write speed.

Action:

Set the length to write bits and the amount of write precompensation to apply.

Entry conditions:

HL contains the length of half a zero bit.
A contains the precompensation to apply.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

The speed supplied is the length of half a zero bit in microseconds. A one bit is written as twice the length of a zero bit. The speed supplied can be related to the average baud rate (assuming equal numbers of ones and zeros) by the following equation:

Average baud rate

= 1 000 000 / (3 * Half zero length)
= 333 333 / Half zero length

The half zero length must lie between 130 and 480 microseconds. Values outside this range will cause read and write errors.

The precompensation supplied is the extra length, in microseconds, to add to half a one bit and to subtract from half a zero bit under certain conditions. The amount of precompensation required varies with the speed (more is required at higher baud rates).

The precompensation may lie between 0 and 255 microseconds although the higher settings are not useful as they will cause read and write errors.

The default half zero length and precompensation settings are 333 microseconds (1000 baud) and 25 microseconds respectively. The commonly used faster setting is 167 microseconds (2000 baud) with 50 microseconds of precompensation. These values have been determined after extensive testing and the user is advised to stick to them.

Related entries:

CAS INITIALISE

121: CAS NOISY #BC6B

Enable or disable prompt messages.

Action:

Disabling messages will prevent the prompt and information messages from being printed. It will not prevent error messages from being printed. Enabling messages allows all messages to be printed.

Entry conditions:

If messages are to be enabled:

> A must be zero.

If messages are to be disabled:

> A must be non-zero.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The prompt and information messages which are turned off are:

> Press PLAY then any key:
> Press REC and PLAY then any key:
> Found <FILENAME> block <N>
> Loading <FILENAME> block <N>
> Saving <FILENAME> block <N>

The error messages which are not turned off are:

> Read error <x>
> Write error a
> Rewind tape

Related entries:

CAS INITIALISE

122: CAS START MOTOR #BC6E

Start the cassette motor.

Action:

Turn the cassette motor on and wait for it to pick up speed if it was previously off.

Entry conditions:

No conditions.

Exit conditions:

If the motor turned on OK:

Carry true.

If the user hit escape:

Carry false.

Always:

A contains the previous motor state.

Other flags corrupt.
All other registers preserved.

Notes:

If the motor is not already on then the routine waits for approximately two seconds to allow the tape to reach full speed.

The motor is always turned on by this routine. If the user hits the escape key then the time spent waiting for the motor to pick up speed is truncated.

The previous motor state may be passed to CAS RESTORE MOTOR.

Related entries:

CAS RESTORE MOTOR
CAS STOP MOTOR

123: CAS STOP MOTOR #BC71

Stop the cassette motor.

Action:

Turn the cassette motor off and return its previous state.

Entry conditions:

No conditions.

Exit conditions:

If the motor was turned off OK:

Carry true.

If the user hit escape:

Carry false.

Always:

A contains the previous motor state.

Other flags corrupt.
All other registers preserved.

Notes:

The motor is always turned off by this routine. There is no delay to allow the motor to slow down.

The previous motor state may be passed to CAS RESTORE MOTOR.

Related entries:

CAS RESTORE MOTOR
CAS START MOTOR

124: CAS RESTORE MOTOR #BC74

Restore previous state of cassette motor.

Action:

Turn the cassette motor on or off again. Wait for motor to pick up speed when turning the motor on if it is currently off.

Entry conditions:

A contains the previous motor state.

Exit conditions:

If the motor was turned on or off OK:

Carry true.

If the user hit escape:

Carry false.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

This routine uses the previous motor state as returned by CAS START MOTOR or CAS STOP MOTOR.

If calling this routine results in the motor being turned on when it is currently off then the routine waits for approximately two seconds to allow the tape to reach full speed.

The motor is always turned on or off (as appropriate) by this routine. If the user hits the escape key then this merely truncates the time spent waiting for the motor to pick up speed.

Related entries:

CAS START MOTOR
CAS STOP MOTOR

125: CAS IN OPEN #BC77

Open a file for input.

Action:

Set up the read stream for reading a file and read the first block.

Entry conditions:

B contains the length of the filename.
HL contains the address of the filename.
DE contains the address of a 2K buffer to use.

Exit conditions:

If the file was opened OK:

> Carry true.
> Zero false.
> HL contains the address of a buffer containing the file header.
> DE contains the data location (from the header).
> BC contains the logical file length (from the header).
> A contains the file type (from the header).

If the stream is in use:

> Carry false.
> Zero false.
> A, BC, DE and HL corrupt.

If the user hit escape:

> Carry false.
> Zero true.
> A, BC, DE and HL corrupt.

Always:

> IX and other flags corrupt.
> All other registers preserved.

Notes:

The 2K buffer (2048 bytes) supplied is used to store the contents of a block of the file when it is read from tape. It will remain in use until the file is closed by calling either CAS IN CLOSE or CAS IN ABANDON. The buffer may lie anywhere in memory, even underneath a ROM.

The filename passed is copied into the read stream descriptor. If it is longer than 16 characters then it is truncated to 16 characters. If it is shorter than 16 characters then it is padded with nulls (#00) to 16 characters. While the filename may contain any character, it is best to avoid nulls. Lower case ASCII letters (characters #61..#7A) are converted to their upper case equivalents (characters #41..#5A). The filename may lie anywhere in RAM, even underneath a ROM.

The filename is normally the name of the file that is to be read. However, a zero length filename (or one starting with a null) is treated specially. It is taken to mean read the next file on the tape.

When the file is opened for reading the first block of the file is read immediately. The address of the area where the header from this block is stored is passed back to the user so that information can be extracted from it. This area will lie in the central 32K of RAM. The user is not allowed to write to the header, only to read from it. The Cassette Manager uses some fields in the header for its own purposes and so these may differ from those read from the tape. The file type, logical length, entry point and all user fields will remain unchanged. (See section 8 for a description of the header.)

Related entries:

CAS IN ABANDON
CAS IN CHAR
CAS IN CLOSE
CAS IN DIRECT
CAS OUT OPEN

126: CAS IN CLOSE #BC7A

Close the input file properly.

Action:

Mark the read stream as closed.

Entry conditions:

No conditions.

Exit conditions:

If the stream was closed OK:

Carry true.

If the stream was not open:

Carry false.

Always:

A, BC, DE, HL and other flags corrupt.
All other registers preserved.

Notes:

This routine should be called to close a file after reading from it using either CAS IN CHAR or CAS IN DIRECT.

The user may reclaim the buffer passed to CAS IN OPEN after calling this routine.

Related entries:

CAS IN ABANDON
CAS IN OPEN
CAS OUT CLOSE

127: CAS IN ABANDON #BC7D

Close the input file immediately.

Action:

Abandon reading from the read stream and close it.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine is intended for use after an error or in similar circumstances.

The user may reclaim the buffer passed to CAS IN OPEN after calling this routine.

Related entries:

CAS IN CLOSE
CAS IN OPEN
CAS OUT ABANDON

128: CAS IN CHAR #BC80

Read a character from the input file.

Action:

Read a character from the input stream. Fetches blocks from tape as required.

Entry conditions:

No conditions.

Exit conditions:

If the character was read OK:

Carry true.
Zero false.
A contains the character read from the file.

If the end of the file was found:

Carry false.
Zero false.
A corrupt.

If the user hit escape:

Carry false.
Zero true.
A corrupt.

Always:

IX and other flags corrupt.
All other registers preserved.

Notes:

If the user has previously pressed escape or the stream is not open as expected then this is reported as the end of the file.

Once the first character has been read from a file it can only be used for character by character access. It is not possible to switch to direct reading (by CAS IN DIRECT).

Related entries:

CAS IN CLOSE
CAS IN DIRECT
CAS IN OPEN
CAS OUT CHAR
CAS RETURN
CAS TEST EOF

134: CAS OUT ABANDON #BC92

Close the output file immediately.

Action:

Abandon the output file and mark the write stream closed. Any unwritten data is discarded and not written to tape.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine is intended for use after an error or in similar circumstances.

Related entries:

CAS IN ABANDON
CAS OUT CLOSE
CAS OUT OPEN

135: CAS OUT CHAR #BC95

Write a character to the output file.

Action:

Add a character to the buffer for the write stream. If the buffer is already full then it is written to tape before the new character is inserted.

Entry conditions:

A contains the character to write.

Exit conditions:

If the character was written OK:

Carry true.
Zero false.

If the file was not open as expected:

Carry false.
Zero false.

If the user hit escape:

Carry false.
Zero true.

Always:

A, IX and other flags corrupt.
All other registers preserved.

Notes:

If this routine returns the file not open as expected condition then either the user has hit escape previously or the file has been written using CAS OUT DIRECT. In either case, or if escape is pressed, the character sent will be discarded.

It is necessary to call CAS OUT CLOSE after sending all the characters to the file to ensure that the last block of the file is written to the tape.

Once this routine has been called it is not possible to switch to directly writing the file.

Related entries:

CAS IN CHAR
CAS OUT CLOSE
CAS OUT DIRECT
CAS OUT OPEN

136: CAS OUT DIRECT #BC98

Write the output file directly from store.

Action:

Write the contents of store directly out to the output file.

Entry conditions:

HL contains the address of the data to write.
DE contains the length of the data to write.
BC contains the entry address (to go into the header).
A contains the file type (to go into the header).

Exit conditions:

If the file was written OK:

Carry true.
Zero false.

If the file was not open as expected:

Carry false.
Zero false.

If the user hit escape:

Carry false.
Zero true.

Always:

A, BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

After writing the file it must be closed using CAS OUT CLOSE to ensure that the last block of the file is written to tape.

It is not possible to change the method for writing files from character output (using CAS OUT CHAR) to direct output (using CAS OUT DIRECT) or vice versa once the method has been chosen. Nor is it possible to directly write a file in two or more parts by calling CAS OUT DIRECT more than once - this will write corrupt data. Attempting to break these rules will result in a file not open as expected error.

Related entries:

CAS IN DIRECT
CAS OUT OPEN
CAS OUT CLOSE

137: CAS CATALOG #BC9B

Generate a catalogue from the tape.

Action:

Read file blocks to check their validity and print information about them on the screen.

Entry conditions:

DE contains the address of a 2K buffer to use.

Exit conditions:

If the cataloguing went OK:

Carry true.
Zero false.

If the read stream was in use:

Carry false.
Zero false.

If an error occurred:

Carry false.
Zero true.

Always:

A, BC,DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

This routine uses the read stream and so the stream must be closed when it is called. The read stream remains closed when this routine exits. The write stream is unaffected by this routine.

The prompt messages are turned on (see CAS NOISY) by this routine.

When cataloguing the Cassette Manager reads a header record, prints information from it and then reads the data record. This cycle repeats until the user hits the escape key. The information printed is as follows:

FILENAME block N T Ok

FILENAME is the name of the file on the tape, or 'Unnamed file' if the filename starts with a null (character #00).

N is the number of the block. Block 1 is normally the first block in a file.

T is a representation of the file type of the file. It is formed by adding #24 (the character '$') to the file type byte masked with #0F (to remove the version number field). The standard file types are thus:

$	a BASIC program file
%	a protected BASIC program file
*	an ASCII text file (default file type)
&	a binary file
'	a protected binary file

Other file types are possible but will not have been written by the BASIC in the on-board ROM. See section 8.4 for a description of the file type byte.

Ok is printed after the end of the data record. This shows that the data was read without errors and also serves to indicate the end of the data on tape (to help avoid over-recording a tape file).

Related entries:

CAS NOISY

138: CAS WRITE #BC9E

Write a record to tape.

Action:

Write a record to the cassette. This routine is used by the higher level routines (CAS OUT CHAR, CAS OUT DIRECT and CAS OUT CLOSE) to write the header and data records that make up a tape file.

Entry conditions:

HL contains the address of the data to write.
DE contains the length of the data to write.
A contains the sync character to write at the end of the leader.

Exit conditions:

If the record was written OK:

> Carry true.
> A corrupt.

If an error occured or the user hit escape:

> Carry false.
> A contains an error code.

Always:

> BC, DE, HL, IX corrupt.
> All other registers preserved.

Notes:

A data length of 0 passed to this routine is taken to mean 65536 bytes and all of memory will be written to tape. (This is unlikely to be useful).

The data to be written may lie anywhere in RAM, even underneath a ROM.

The sync character is used to distinguish header records (sync is #2C) from data records (sync is #16). Other sync characters could be used but the resulting record would require special action to be taken to read it.

The error codes returned by this routine are:

0	Break	The user hit the escape key.
1	Overrun	The Cassette Manager was unable to get back to writing a bit fast enough.

Because reading and writing the tape requires stringent timing considerations interrupts are disabled whilst the tape is being written (potentially a period of over 5 minutes). It would be unpleasant to have the sound chip making a nois for all this time so the Sound Manager is shut down (SOUND RESET). When writing to the tape has finished interrupts are re-enabled.

The cassette motor is started by this routine (in case it is not already on) and restored to its previous state when writing is completed.

Related entries:

CAS CHECK
CAS READ

139: CAS READ #BCA1

Read a record from tape.

Action:

Read a whole or part record from the cassette. This routine is used by the higher level routines (CAS IN CHAR, CAS IN DIRECT and CAS CATALOG amongst others) to read the header and data records that make up a file.

Entry conditions:

HL contains the address to put the data read.
DE contains the length of the data to read.
A contains the sync character expected at the end of the leader.

Exit conditions:

If record was read OK:

> Carry true.
> A corrupt.

If an error occured or the user hit escape:

> Carry false.
> A contains an error code.

Always:

> BC, DE, HL, IX and other flags corrupt.
> All other registers preserved.

Notes:

A data length of 0 passed to this routine is taken to mean 65536 bytes. (This is not useful).

It is not necessary to read the whole of a record from tape. If the length passed is less than the actual length of the record then only that number of bytes will be read. Trying to read more bytes from a record than were written will produce an error, usually an overflow error (see below).

The sync character is used to distinguish header records (sync is #2C) from data records (sync is #16). Other sync characters could be used if the record was written that way.

The error codes returned by this routine are:

0	Break	The user hit the escape key.
1	Overflow	The Cassette Manager found a bit that was too long to read.
2	CRC	A CRC failure was detected.

The cassette motor is started by this routine (in case it is not already on) and restored to its previous state when reading is completed.

Because reading the tape requires stringent timing considerations, interrupts are disabled whilst the tape is being read (potentially a period of over 5 minutes). It would be unpleasant to have the sound chip making a noise for all this time so the Sound Manager is shut down (SOUND RESET). When reading from the tape has finished interrupts are reenabled.

Related entries:

CAS CHECK
CAS WRITE

140: CAS CHECK #BCA4

Compare a record on tape with the contents of store.

Action:

Check that a tape record contains a correct version of the data supplied. This routine is intended to be used after writing records to check that they were written correctly.

Entry conditions:

HL contains the address of the data to check.
DE contains the length of the data to check.
A contains the sync character expected at the end of the leader.

Exit conditions:

If the record checked OK:

Carry true.
A corrupt.

If an error occured or the user hit escape:

Carry false.
A contains an error code.

Always:

BC, DE, HL, IX and other flags corrupt.
All other registers preserved.

Notes:

A data length of 0 passed to this routine is taken to mean 65536 bytes. (This is bound to produce a check failure).

It is not necessary to check the whole of a record on tape. If the length passed is less than the actual length of the record then only that number of bytes will be checked. Trying to check more bytes in a record than were written will produce an error of some sort (see below).

The data to be checked may lie anywhere in RAM, even underneath a ROM.

The sync character is used to distinguish header records (sync is #2C) from data records (sync is #16). Other sync characters could be used.

The error codes returned by this routine are:

0	Break	The user hit the escape key.
1	Overrun	The Cassette Manager found a bit that was too long to read.
2	CRC	A CRC failure was detected.
3	Different	The data read from tape did not agree with that in memory.

The cassette motor is started by this routine (in case it is not already on) and restored to its previous state when checking is completed.

Because reading from the tape requires stringent timing considerations, interrupts are disabled whilst the tape is being checked (potentially a period of over 5 minutes). It would be unpleasant to have the sound chip making a noise for all this time so the Sound Manager is shut down (SOUND RESET). When checking has finished interrupts are reenabled.

Related entries:

CAS READ
CAS WRITE

141: SOUND RESET #BCA7

Reset the Sound Manager.

Action:

Re-initialise the Sound Manager - shut the sound chip up and clear all queues.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The sound queues are cleared.
Any current sound is stopped.
The sound generator chip is silenced.

This routine enables interrupts.

Related entries:

SOUND HOLD

142: SOUND QUEUE

Add a sound to a sound queue.

Action:

Try to add a sound to the sound queue of one or more channels. If the sound queue of any of the channels is full then no sound will be issued to any channel.

Entry conditions:

HL contains the address of a sound program which must lie in the central 32K of RAM.

Exit conditions:

If the sound was added to the queue(s):

Carry true.
HL corrupt.

If at least one queue was full:

Carry false.
HL preserved.

Always:

A, BC, DE, IX and other flags corrupt.
All other registers preserved.

Notes:

The sound program is laid out as follows:

Byte 0:	Channels to use and rendezvous requirements.
Byte 1:	Amplitude envelope to use.
Byte 2:	Tone envelope to use.
Bytes 3..4:	Tone period.
Byte 5:	Noise period.
Byte 6:	Initial amplitude.
Bytes 7..8:	Duration or envelope repeat count.

All values in the sound program are masked into the appropriate range before being used.

The channels to issue the sound on are encoded into byte 0 as follows:

Bit 0:	Issue on channel A.
Bit 1:	Issue on channel B.
Bit 2:	Issue on channel C.

Bit 3: Rendezvous with channel A.
Bit 4: Rendezvous with channel B.
Bit 5: Rendezvous with channel C.
Bit 6: Hold until released.
Bit 7: Flush queue

A channel will ignore an order to rendezvous with itself. Sounds issued on multiple channels implicitly rendezvous with each other. Sounds that are ordered to rendezvous will be issued to the sound generator starting at the same time.

Setting the hold bit prevents the sound from running until it is released by calling SOUND RELEASE (or a routine having a similar effect). Setting the flush bit will empty the queue and abandon any currently active sound thus allowing the new sound to start immediately.

The amplitude envelope is in the range 0..15. Envelopes 1..15 are the amplitude envelopes that can be set using SOUND AMPL ENVELOPE. Envelope 0 means use no amplitude envelope, simply hold the initial amplitude for 2 seconds or the duration specified.

The tone envelope is in the range 0..15. Envelopes 1..15 are the tone envelopes that can be set using SOUND TONE ENVELOPE. Envelope 0 means use no tone envelope, simply hold the initial tone.

A tone period of 0 means do not generate any tone. Tone periods in the range 1..4095 specify the period of the tone in 8 microsecond units.

The noise period is in the range 0..31. Noise periods 1..31 specify the period of the noise component of a sound. A noise period of 0 means use no noise.

The initial amplitude is in the range 0..15. Amplitude 0 being no initial sound, amplitude 15 being maximum volume.

Bytes 7 and 8 store the sound time. If this is zero then the amplitude envelope is obeyed once. If the sound time is negative then the amplitude envelope is obeyed minus the sound time number of times (i.e. 1..32768 times). If the sound time is positive but not zero then it is taken to be the duration of the sound in 1/100s of a second.

If a duration is specified when an amplitude envelope is in use then the duration given sets the length of the sound. If the duration is shorter than the envelope then the envelope is truncated. If the duration is longer than the envelope then the final amplitude of the envelope is sustained until the duration expires. Tone envelopes are treated in much the same way as amplitude envelopes except that they never specify the length of the sound.

The sound event that is run when a sound queue has a free slot is disarmed on the channels specified in this command.

All sounds currently held by SOUND HOLD are automatically released when this routine is called. Also, the sound queue event is disarmed (see SOUND ARM EVENT).

SOUND QUEUE may enable interrupts.

Related entries:

SOUND ARM EVENT
SOUND CHECK
SOUND RELEASE

143: SOUND CHECK #BCAD

Ask if there is space in a sound queue.

Action:

Ask the status of a sound channel. The status includes the number of free spaces in the sound queue and whether the channel is held.

Entry conditions:

A contains the bit for the channel to test.

Exit conditions:

A contains the channel status.

BC, DE, HL and flags corrupt.
All other registers preserved.

Notes:

The channel to ask the status of is encoded as follows:

Bit 0:	Ask about channel A.
Bit 1:	Ask about channel B.
Bit 2:	Ask about channel C.

If more than one bit is set then the status of only one channel is returned. The channels are tested in the order given above.

The status returned is encoded as follows:

Bits 0..2:	Contain the number of free slots in the channel's sound queue.
Bit 3:	The channel is awaiting a rendezvous with channel A.
Bit 4:	The channel is awaiting a rendezvous with channel B.
Bit 5:	The channel is awaiting a rendezvous with channel C.
Bit 6:	The channel is held.
Bit 7:	The channel is active (producing a sound).

Calling this routine disarms the sound queue event that occurs when the queue has a free slot for the channel returned (see SOUND ARM EVENT).

This routine may enable interrupts.

Related entries:

SOUND ARM EVENT
SOUND QUEUE

144: SOUND ARM EVENT #BCB0

Set up an event to be run when a sound queue becomes empty.

Action:

Arm the sound event to be run when a free slot occurs in a channel's sound queue.

Entry conditions:

A contains the bit for the channel to arm.
HL contains the address of an event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The channel for which to arm the event is encoded as follows:

Bit 0:	Arm channel A.
Bit 1:	Arm channel B.
Bit 2:	Arm channel C.

If more than one bit is set then only one channel is armed. The channels are armed in the order given above.

The event block passed must be initialised (by KL INIT EVENT).

The event will be 'kicked' when a free slot occurs in the queue. If there is a free slot in the queue when this routine is called then the event will be 'kicked' immediately.

The sound event is disarmed automatically when SOUND QUEUE or SOUND CHECK is called. It is also disarmed when the event is run. Thus, the event routine will need to rearm the sound event to keep it running continuously.

This routine may enable interrupts.

Related entries:

KL INIT EVENT
SOUND CHECK
SOUND QUEUE

145: SOUND RELEASE #BCB3

Allow sounds which are individually held to start.

Action:

Release held sounds on a number of channels. This allows sounds that were marked with a hold bit when they were set up by SOUND QUEUE to start (other factors willing).

Entry conditions:

A contains bits for the channels to release.

Exit conditions:

AF, BC, DE, HL and IX corrupt.
All other registers preserved.

Notes:

The channels to release are encoded as follows:

Bit 0:	Release channel A.
Bit 1:	Release channel B.
Bit 2:	Release channel C.

All channels that are specified are released.

All sounds currently held by SOUND HOLD are automatically released.

This routine may enable interrupts.

Related entries:

SOUND QUEUE

146: SOUND HOLD #BCB6

Stop all sounds in midflight.

Action:

This stops all sounds immediately. The sounds can be started again by calling SOUND CONTINUE.

Entry conditions:

No conditions.

Exit conditions:

If a sound was active:

Carry true.

If no sound was active:

Carry false.

Always:

A, BC, HL and other flags corrupt.
All other registers preserved.

Notes:

Sounds that are held by this routine are automatically restarted when SOUND QUEUE or SOUND RELEASE are called as well as when SOUND CONTINUE itself is called.

The sound is stopped by halting the execution of sound and tone envelopes and setting the sound chip volume to zero for all channels. When the sound is restarted it will continue from as near where it was stopped as is possible.

This routine enables interrupts.

Related entries:

SOUND CONTINUE
SOUND RESET

147: SOUND CONTINUE #BCB9

Restart sounds after they have all been held.

Action:

Allow sounds that have been held by calling SOUND HOLD to continue.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and IX corrupt.
All other registers preserved.

Notes:

If no sounds are held then no action is taken.

This routine may enable interrupts.

Related entries:

SOUND HOLD
SOUND RELEASE

148: SOUND AMPL ENVELOPE #BCBC

Set up an amplitude envelope.

Action:

Set up one of the 15 user programmable amplitude (volume) envelopes.

Entry conditions:

A contains an envelope number.
HL contains the address of an amplitude data block.

Exit conditions:

If envelope has been set up OK:

Carry true.
HL contains the address of the data block + 16.
A and BC corrupt.

If envelope number is invalid:

Carry false.
A, B, and HL preserved.

Always:

DE and other flags corrupt.
All other registers preserved.

Notes:

The envelope to set up is specified by a number in the range 1..15. No envelope is set up if a number outside this range is passed.

The amplitude data block is copied into the amplitude envelope. The data block may lie in ROM or in RAM. It may not lie in RAM hidden underneath a ROM.

The amplitude data block has the following layout:

Byte 0:	Count of sections in the envelope.
Bytes 1..3:	First section of the envelope.
Bytes 4..6:	Second section of the envelope.
Bytes 7..9:	Third section of the envelope.
Bytes 10..12:	Fourth section of the envelope.
Bytes 13..15:	Fifth section of the envelope.

The first byte of the amplitude data block specifies the number of sections used in the envelope. Sections not used need not be set up. An envelope using no sections has a special meaning - hold a constant volume lasting for 2 seconds.

The number of sections to use is not checked, if a number outside the range 0..5 is supplied then this will have unpredictable effects. This should be avoided.

Each section of the amplitude data block can specify either a hardware or a software envelope. This is indicated by the first byte of the section.

A software envelope section is laid out as follows:

Byte 0:	Step count.
Byte 1:	Step size.
Byte 2:	Pause time.

The fact that this is a software envelope section rather than a hardware envelope section is indicated by byte 0 not having bit 7 set.

If the step count is in the range 1..127 then the step size is added to the volume that number of times with a wait equal to the pause time in 1/100s of a second after each addition.

If the step count is 0 the the step size is taken to be an absolute volume setting. A single wait of the pause time in 1/100s of a second is made.

After calculating the new volume this is masked with #0F to make sure it is legal. Thus, all arithmetic on the volume is carried out modulo 16.

A pause time of 0 is taken to mean 256 1/100s of a second.

A hardware envelope section is laid out as follows:

Byte 0:	Envelope shape.
Byte 1..2:	Envelope period.

The fact that this is a hardware envelope section rather than a software envelope section is indicated by byte 0 having bit 7 set.

The envelope shape (masked with #7F) is sent to register 13 of the sound generator. This sets the shape of the hardware envelope and whether it repeats (see Appendix IX for details).

The envelope period is sent to registers 11 and 12 of the sound generator. These set the the length of the hardware envelope (see Appendix IX for details).

The section after a hardware section should be a pause long enough to allow the hardware envelope to operate. A pause can be constructed using a software envelope with a step size of 0 and with the repeat count and pause time juggled to give the right total time.

There is no protection against changing an envelope whilst it is in use. This could have unpredictable effects and should be avoided.

The length of the sound can either be determined by the duration supplied when the sound is queued or by the envelope terminating (see SOUND QUEUE). If a duration is set that is shorter than the envelope then the envelope is truncated. If the duration is longer than the envelope then the final volume is sustained until the duration expires.

Related entries:

SOUND A ADDRESS
SOUND TONE ENVELOPE

149: SOUND TONE ENVELOPE #BCBF

Set up a tone envelope.

Action:

Set up one of the 15 user programmable tone envelopes.

Entry conditions:

A contains an envelope number.
HL contains the address of a tone data block.

Exit conditions:

If the envelope has been set up OK:

> Carry true.
> HL contains the address of the data block + 16.
> A and BC corrupt.

If the envelope number is invalid:

> Carry false.
> A, BC and HL preserved.

Always:

> DE and other flags corrupt.
> All other registers preserved.

Notes:

The envelope to set up is specified by a number in the range 1..15. No envelope is set up if a number outside this range is passed.

The tone data block is copied into the tone envelope. The data block may lie in ROM or in RAM. It may not lie in RAM hidden underneath a ROM.

The tone data block has the following layout:

Byte 0:	Count of sections in the envelope.
Bytes 1..3:	First section of the envelope.
Bytes 4..6:	Second section of the envelope.
Bytes 7..9:	Third section of the envelope.
Bytes 10..12:	Fourth section of the envelope.
Bytes 13..15:	Fifth section of the envelope.

The first byte of the tone data block (masked with #7F) specifies the number of sections used in the envelope. Sections not used need not be set up. An envelope using no sections will not alter the tone (i.e. no enveloping). The number of sections to use is not checked, if a number outside the range 0..5 is supplied then this will have unpredictable effects. This should be avoided.

The top bit, bit 7, of the count is used to indicate a repeating envelope. If this bit is set then when the last section of the envelope finishes the first will be used again.

Each section of the tone data block is laid out as follows:

Byte 0:	Step count.
Byte 1:	Step size.
Byte 2:	Pause time.

If the step count lies in the range #00..#EF then the section is a relative section. The step size is sign extended (bit 7 is copied to bits 8..15) and is added to the current tone period the number of times specified by the step count. After each addition a wait of the pause time in 1/100s of a second is made. The sound chip only uses the lower 12 bits of the tone period so all arithmetic is carried out modulo #1000.

A step count of 0 is taken to mean 1 step whilst a pause time of 0 is taken to mean 256 1/100s of a second.

If the step count lies in the range #F0..#FF then the section is an absolute section. The least significant four bits of the step count are taken to be the most significant byte of the tone period and the step size is taken to be the least significant byte. This tone period is set immediately and is followed by a pause whose length is set by the pause time in 1/100s of a second.

There is no protection against changing an envelope whilst it is in use. This could have unpredictable effects and should be avoided.

If the tone envelope finishes before the end of the sound (as set when the sound was queued) then the final tone is held. i.e. The tone envelope does not affect the length of the sound.

Related entries:

SOUND AMPL ENVELOPE
SOUND T ADDRESS

150: SOUND A ADDRESS #BCC2

Get the address of an amplitude envelope.

Action:

Ask where the data area for an amplitude envelope is stored.

Entry conditions:

A contains an envelope number.

Exit conditions:

If the envelope was found OK:

Carry true.
HL contains the address of the amplitude envelope.
BC contains the length of an envelope (16 bytes).

If the envelope number was invalid:

Carry false.
HL corrupt.
BC preserved.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

The envelope number must lie in the range 1..15.

The amplitude envelope is laid out as described in SOUND AMPL ENVELOPE.

Related entries:

SOUND AMPL ENVELOPE
SOUND T ADDRESS

151: SOUND T ADDRESS #BCC5

Get the address of a tone envelope.

Action:

Ask where the data area for a tone envelope is stored.

Entry conditions:

A contains an envelope number.

Exit conditions:

If the envelope was found OK:

Carry true.
HL contains the address of the tone envelope.
BC contains the length of an envelope (16 bytes).

If the envelope number was invalid:

Carry false.
HL corrupt.
BC preserved.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

The envelope number must lie in the range 1..15.

The tone envelope is laid out as described in SOUND TONE ENVELOPE.

Related entries:

SOUND A ADDRESS
SOUND TONE ENVELOPE

152: KL CHOKE OFF #BCC8

Reset the Kernel - clears all event queues etc.

Action:

This entry completely clears all event queues, the various timer and frame flyback lists and so on. The effect is to dispose of any pending synchronous events and to halt all timer related functions other than sound generation and keyboard scanning.

Entry conditions:

No conditions.

Exit conditions:

B contains the ROM select address of the current foreground ROM (if any).
DE contains the address at which the current foreground ROM was entered.
C contains the ROM select address for a RAM foreground program.

AF and HL corrupt.
All other registers preserved.

Notes:

If the current foreground program is in RAM then the ROM select address and entry point returned are both zero. i.e. The default ROM (ROM 0) at its entry address.

KL CHOKE OFF forms part of the close down required before a new RAM foreground program is loaded, as is required by MC BOOT PROGRAM.

The close down must ensure that there are no interrupt or other events active and using memory which might be damaged by loading a new program into memory. In the complete close down MC BOOT PROGRAM does:

SOUND RESET	to kill off sound generation
an OUT to I/O port #F8FF	to reset any external interrupt sources.
KL CHOKE OFF	to kill off events etc.
KM RESET	to reset any keyboard indirections and the break event.
TXT RESET	to reset any Text VDU indirections.
SCR RESET	to reset any screen indirections.

The values returned by KL CHOKE OFF are used by MC BOOT PROGRAM if the program load fails.

This information is included for the reader's interest. MC BOOT PROGRAM is the recommended means of loading and entering a RAM foreground program. MC START PROGRAM is the recommended means of entering a ROM foreground program, or a RAM foreground program which has already been loaded.

KL CHOKE OFF disables interrupts.

Related entries:

MC BOOT PROGRAM
MC START PROGRAM

153: KL ROM WALK #BCCB

Find and initialise all background ROMs.

Action:

Background ROMs provide support for expansion hardware or augment the software facilities of the machine. If the facilities provided by the background ROMs are to be available, the foreground program must initialise them. This routine finds and initialises all background ROMs.

Entry conditions:

DE contains address of the first usable byte of memory (lowest address).
HL contains address of the last usable byte of memory (highest address).

Exit conditions:

DE contains the address of the new first usable byte of memory.
HL contains the address of the new last usable byte of memory.

AF and BC corrupt.
All other registers preserved.

Notes:

When a foreground program is entered it is passed the addresses of the first and last bytes in memory which it may use. The area of memory outside this is used to store firmware variables, the stack, the jumpblocks and the screen memory. From the area available for a foreground program to use, the areas for background programs to use must be allocated.

The foreground program should initialise background ROMs at an early stage, before it uses the memory it is given. It may choose whether to enable background ROMs or not. KL INIT BACK may be used to initialise a particular background ROM or this routine may be used to initialise all available background ROMs.

KL ROM WALK inspects the ROMs at ROM select addresses in the range 1..7. The power-up initialisation entry of each background ROM found is called. This entry may allocate some memory for the background ROM's use by adjusting the values in DE and HL before returning. Once the ROM has been initialised the Kernel adds it to the list of external command servers, and notes the base of the area which the ROM has allocated to itself at the top of memory (if any). Subsequent FAR CALLs to entries in the ROM will automatically set the IY index register to point at the ROM's upper memory area.

See section 9.4 for a full description of background ROMs.

Related entries:

KL FIND COMMAND
KL INIT BACK
KL LOG EXT

154: KL INIT BACK #BCCE

Initialise a particular background ROM.

Action:

Background ROMs provide support for expansion hardware or augment the software facilities of the machine. If the facilities provided by the background ROMs are to be available the foreground program must initialise them. This routine selects and initialises a particular background ROM.

Entry conditions:

C contains the ROM select address of the ROM to initialise.
DE contains address of the first usable byte of memory (lowest address).
HL contains address of the last usable byte of memory (highest address).

Exit conditions:

DE contains the address of the new first usable byte of memory.
HL contains the address of the new last usable byte of memory.

AF and B corrupt.
All other registers preserved.

Notes:

The ROM select address must be in the range 1..7 or the request will be ignored. The ROM at this address must be a background ROM or the request will be ignored.

When a foreground program is entered it is passed the addresses of the first and last bytes in memory which it may use. The area of memory outside this is used to store firmware variables, the stack, the jumpblocks and the screen memory. From the area available for a foreground program to use, the areas for background programs to use must be allocated.

The foreground program should initialise background ROMs at an early stage, before it uses the memory it is given. It may choose whether to enable background ROMs or not. KL ROM WALK may be used to initialise all available ROMs or this routine may be used to initialise particular ROMs.

This routine causes the background ROM's power-up initialisation entry to be called. This entry may allocate some memory for the background ROM's use by adjusting the values in DE and HL before returning. Once the ROM has been initialised the Kernel adds it to the list of external command servers, and notes the base of the area which the ROM has allocated to itself at the top of memory (if any). Subsequent FAR CALLs to entries in the ROM will automatically set the IY index register to point at the ROM's upper memory area.

155: KL LOG EXT #BCD1

Introduce an RSX to the Firmware.

Action:

RSXs (Resident System Extensions) are similar to background ROMs, but are loaded into RAM. This routine must be called to include the RSX on the Kernel's list of external command servers.

Entry conditions:

BC contains the address of the RSX's command table.
HL contains the address of a 4 byte area of RAM for the Kernel's use.

Exit conditions:

DE corrupt.
All other registers preserved.

Notes:

Both the RSX's command table and the Kernel's storage area must lie in the central 32K of memory, i.e. not under a ROM.

The format of a command table is described in section 9.2 and RSXs are discussed in section 9.5.

Related entries:

KL FIND COMMAND
KL INIT BACK

156: KL FIND COMMAND #BCD4

Search for an RSX, background ROM or foreground ROM to process a command.

Action:

All expansion ROMs and RSXs have command tables of the same form. This routine searches all RSXs and background ROMs on the Kernel's list of external command servers looking for a match for the given command name. If the name is found, then the 'far address' of the associated routine is returned. If the command is not a background or RSX command then all the foreground ROMs that can be found are searched for a foreground program with the given name. If a foreground program is found then the system immediately enters it.

Entry conditions:

HL contains the address of the command name to search for.

Exit conditions:

If an RSX or background ROM command was found:

> Carry true.
> C contains the ROM select address.
> HL contains the address of the routine.

If the command was not found:

> Carry false.
> C and HL corrupt.

Always

> A, B and DE corrupt.
> All other registers preserved.

Notes:

The command name passed must be in RAM but may lie underneath a ROM. The name may be any number of characters long but only the first 16 characters are significant. All alphabetic characters in the name should be in upper case and the last character of the name should have bit 7 set.

The ROM select and routine addresses are suitable for calling KL FAR PCHL.

The list of external command servers is generated as background ROMs and RSXs are initialised (see KL ROM WALK, KL INIT BACK and KL LOG EXT). The command tables are scanned in the opposite order to that in which the command servers were introduced. Thus, RSXs will tend to take precedence over background ROMs, since RSX's are, in general, initialised after background ROMs. Background ROMs are normally initialised in reverse order of ROM select address, so lower numbered ROMS will take precedence over higher.

See section 9.2 for a full description of the format of expansion ROM command tables.

The first entry in a background ROM's command name table (the one associated with the power-up entry) may be used as the ROM's name. KL FIND COMMAND may be used, therefore, to find out whether a particular background ROM has been initialised.

When searching for a foreground program, ROMs are inspected starting with ROM 0 and working up until an unused ROM address is found.

The on-board BASIC may be entered by searching for and invoking the command BASIC.

If a foreground ROM command is found the ROM is entered unconditionally and this routine never returns.

Related entries:

KL INIT BACK
KL LOG EXT
KL ROM WALK
MC START PROGRAM

157: KL NEW FRAME FLY #BCD7

Initialise and put a block onto the frame flyback list.

Action:

The Kernel maintains a list of events to be kicked each time frame flyback occurs. This routine initialises a block and adds it to the list.

Entry conditions:

HL contains the address of the frame flyback block.
B contains the event class.
C contains the ROM select address of the event routine.
DE contains the address of the event routine.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The frame flyback block is 9 bytes long and must lie in the central 32K of RAM. The last 7 bytes of the frame flyback block are an event block which is initialised to reflect the parameters passed in B, C and DE (see KL INIT EVENT). The exact layout of a frame flyback block is described in Appendix X.

The frame flyback block is appended to the frame flyback list if it is not already on it.

This routine enables interrupts.

Related entries:

KL ADD FRAME FLY
KL DEL FRAME FLY
KL INIT EVENT

158: KL ADD FRAME FLY #BCDA

Put a block onto the frame flyback list.

Action:

The Kernel maintains a list of events to be kicked each time frame flyback occurs. This routine adds a block to the list.

Entry conditions:

HL contains the address of the frame flyback block.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The frame flyback block is 9 bytes long and it must lie in the central 32K of RAM. The last 7 bytes of the frame flyback block are an event block which must be initialised separately before calling this routine. The exact layout of a frame flyback block is described in Appendix X.

The block is appended to the frame flyback list if it is not already on it.

This routine enables interrupts.

Related entries:

KL DEL FRAME FLY
KL INIT EVENT
KL NEW FRAME FLY

159: KL DEL FRAME FLY #BCDD

Remove a block from the frame flyback list.

Action:

The Kernel maintains a list of events to be kicked each time frame flyback occurs. This routine removes a block from the list.

Entry conditions:

HL contains the address of the frame flyback block.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

This routine does nothing if the block is not on the list.

Removing a block from the list only prevents the event being kicked again. It does not affect any outstanding frame flyback events.

This routine enables interrupts.

Related entries:

KL ADD FRAME FLY
KL NEW FRAME FLY

160: KL NEW FAST TICKER #BCE0

Initialise and put a block onto the fast ticker list.

Action:

The Kernel maintains a list of events to be kicked each time the 1/300th of a second timer interrupt occurs. This is known as the fast ticker list. This routine initialises a block and adds it to the list.

Entry conditions:

HL contains the address of the fast ticker block.
B contains the event class.
C contains the ROM select address of the event routine.
DE contains the address of the event routine.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The fast ticker block is 9 bytes long and must lie in the central 32K of RAM. The last 7 bytes of the fast ticker block are an event block which is initialised to reflect the parameters passed in B, C and DE (see KL INIT EVENT). The exact layout of a fast ticker block is described in Appendix X.

The fast ticker block is appended to the fast ticker list if it is not already on it.

The fast ticker facility is not intended for general use. However, it does allow relatively short times to be measured giving greater resolution than the general ticker facilities.

This routine enables interrupts.

Related entries:

KL ADD FAST TICKER
KL ADD TICKER
KL DEL FAST TICKER
KL INIT EVENT
KL TIME PLEASE

161: KL ADD FAST TICKER #BCE3

Put a block onto the fast ticker list.

Action:

The Kernel maintains a list of events to be kicked each time the 1/300th of a second timer interrupt occurs. This is known as the fast ticker list. This routine adds a block to the list.

Entry conditions:

HL contains the address of the fast ticker block.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

The fast ticker block is 9 bytes long and must lie in the central 32K of RAM. The last 7 bytes of the fast ticker block are an event block which must be initialised before calling this routine. The exact layout of a fast ticker block is described in Appendix X.

The fast ticker block is appended to the fast ticker list if it is not already on it.

The fast ticker facility is not intended for general use. However, it does allow relatively short times to be measured giving greater resolution than the general ticker facilities.

This routine enables interrupts.

Related entries:

KL ADD TICKER
KL DEL FAST TICKER
KL INIT EVENT
KL NEW FAST TICKER
KL TIME PLEASE

162: KL DEL FAST TICKER #BCE6

Remove a block from the fast ticker list.

Action:

The Kernel maintains a list of events to be kicked each time the 1/300th of a second timer interrupt occurs. This is known as the fast ticker list. This routine removes a block from the list.

Entry conditions:

HL contains the address of the fast ticker block.

Exit conditions:

AF, DE and HL corrupt.
All other registers preserved.

Notes:

This routine does nothing if the block is not on the list.

Removing a block from the list only prevents the event from being kicked again. It does not affect any outstanding fast ticker events.

This routine enables interrupts.

Related entries:

KL ADD FAST TICKER
KL DEL TICKER
KL NEW FAST TICKER

163: KL ADD TICKER #BCE9

Put a block onto the tick list.

Action:

The general purpose timing facility measures time in 1/50th of a second units. The Kernel maintains a list of tick blocks each of which contains a count and a recharge value. Every 1/50th of a second the Kernel processes all the tick blocks, decrementing the count entry of each. If the count entry of a block becomes zero the event contained in the block is 'kicked', and the count is set to the recharge value.

Entry conditions:

HL contains the address of the tick block.
DE contains the initial value for the count entry.
BC contains the value for the recharge entry.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The tick block is 13 bytes long and must lie in the central 32K of memory. The last 7 bytes of the tick block are an event block which must be initialised before this routine is called. The exact layout of a tick block is described in Appendix X.

The count and recharge entries in the block are set. The block is then appended to the tick list if it is not already on the list. This routine may be used, therefore, to change the count and recharge entries of an existing block.

Blocks with a count entry of zero are ignored when the list is processed. Setting a recharge value of zero, therefore, sets up the block as a 'one shot timer'. Since it takes the Kernel time to ignore a tick block, any redundant blocks should be removed from the list as soon as possible.

It is not possible to predict, particularly with synchronous events, how long it will be after the 'kick' before the event routine is actually called. Notwithstanding these delays, the ticker may be used to obtain an exact number of 'kicks' in a given period since the recharge mechanism immediately resets the count. The event counting mechanism will ensure that 'kicks' are not missed, provided that there are never more than 127 outstanding at once.

This routine enables interrupts.

Related entries:

KL ADD FAST TICKER
KL DEL TICKER
KL INIT EVENT

164: KL DEL TICKER #BCEC

Remove block from the tick list.

Action:

If the given block is on the tick list it is removed. The contents of the block are not affected.

Entry conditions:

HL contains the address of the tick block.

Exit conditions:

If the tick block was found on the tick list:

Carry true.
DE contains the count remaining before the next event.

If the tick block was not found on the tick list:

Carry false.
DE corrupt.

Always:

A, HL and other flags corrupt.
All other registers preserved.

Notes:

The contents of the block are not affected by removing it from the list. In particular the continued processing of outstanding events is not affected. The block could be put back on the list at a later date and it could continue counting where it left off.

This routine enables interrupts.

Related entries:

KL ADD TICKER
KL DEL FAST TICKER

165: KL INIT EVENT #BCEF

Initialise an event block.

Action:

Initialise all entries in an event block.

Entry conditions:

HL contains the address of the event block.
B contains the event class.
C contains the ROM select address of the event routine.
DE contains the address of the event routine

Exit conditions:

HL contains the address of the event block +7.

All other registers preserved.

Notes:

The event block is 7 bytes long and must lie in the central 32K of RAM. The layout of an event block is described in Appendix X. See section 11 for a general discussion of events.

The ROM select and address of the routine are the 'far address' of the event routine (see section 2).

The event class is bit significant as follows:

Bit 0:	Near address.
Bits 1..4:	Synchronous event priority.
Bit 5:	Must be zero.
Bit 6:	Express event.
Bit 7:	Asynchronous event.

If the asynchronous bit is set then the event is an asynchronous event, otherwise it is a synchronous event. Asynchronous events do not have priorities and so the priority field is ignored.

If the express event bit is set then the event is an express event. The meaning of this depends on whether the event is synchronous or asynchronous.

All express synchronous events have higher priorities than any normal synchronous event. The priority of a synchronous event is encoded in bits 1..4 of the class, the higher the number the greater the priority. No event may have priority 0. The processing of normal synchronous events may be disabled (by calling KL EVENT DISABLE), while that of express synchronous events may not.

An express asynchronous event will have its event routine called directly from the interrupt path. A normal asynchronous event is processed just before returning from the interrupt (with interrupts are enabled).

If the near address bit is set then the event routine is located either in the lower ROM or in the central 32K of RAM. The ROM select address is ignored and the routine is called directly, rather than through the FAR CALL mechanism, thus reducing the event processing overhead. Where possible asynchronous events should be at 'near addresses'. Express asynchronous events must always be at 'near addresses'.

Event blocks appear in various other blocks handled by the Kernel, including frame flyback, fast ticker and tick blocks. This routine is used to initialise the event block parts of these.

The bytes after the last byte of the event block, even where the block forms part of another block, are not used by the Kernel. When the event routine is called the address of the block is passed to it, so the user may append further information about the event to the block. This allows several similar events to share the same event routine, each event having its 'own' variables appended to its event block.

The event routine has the following entry and exit conditions:

Entry:

If the event routine is at a 'far address':

> HL contains the address of byte 5 of the event block
> (so any appended data can start at address HL+2).

If the event routine is at a 'near address':

> HL contains the address of byte 6 of the event block
> (so any appended data can start at address HL+1).

Exit:

> AF, BC, DE and HL corrupt.
> All other registers preserved.

The event routine may use the IX and IY registers but must preserve them. It may not use the second register set. Express asynchronous events may not enable interrupts.

KL INIT EVENT enables interrupts.

Related entries:

KL DEL SYNCHRONOUS
KL DISARM EVENT
KL EVENT
KL NEW FAST TICKER
KL NEW FRAME FLY
KL NEW TICKER
KL SYNC RESET

166: KL EVENT #BCF2

'Kick' an event block.

Action:

The event mechanism arranges that an event routine be called in response to each 'kick' of an event block. KL EVENT performs the 'kick'.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Unlike the vast majority of Kernel routines this routine may be called from the interrupt path. Because the LOW JUMP instruction in the main firmware jumpblock enables interrupts the user must pick the address part of the 'low address' out of the jumpblock and mask off the top two bits to extract the address in the lower ROM of KL EVENT. The following code does this:

```
LD   DE,(#BCF2+1)       ; extract address part of LOW JUMP
RES  7,D                ; remove upper ROM state from 'low address'
RES  6,D                ; remove lower ROM state from 'low address'
CALL PCDE_INSTRUCTION   ; CALL KL EVENT
```

(If the user is going to perform this operation repeatedly it is recommended that the address should be extracted once and should be stored somewhere).

The effect of the 'kick' depends on the event count in the event block:

Count < 0: The event is disarmed, and kicking it has no effect.

Count > 0: There are other kicks outstanding and the event is being processed. This kick simply increments the count (unless it has already reached the maximum of 127). Once event processing has begun it continues until the count becomes zero or the event is disarmed.

Count = 0: The event is armed but event processing is not active.
The count is incremented and event processing initiated.

How event processing is initiated depends on the event class.

Synchronous Events.

Synchronous events are added to the synchronous event queue in priority order. It is the responsibility of the foreground program to process the synchronous event queue regularly.

Synchronous event routines are called when the foreground program calls KL DO SYNC, the event count is then dealt with when KL DONE SYNC is called.

Asynchronous Events.

a. Not in the Interrupt Path

The event routine is called immediately. When the routine returns, if the event count greater than zero it is decremented. If the count is still greater than zero then the procedure is repeated.

b. In the Interrupt Path - Normal Asynchronous Event

The event is placed on the interrupt event pending queue. On exit from the interrupt path the Kernel processes all events on the interrupt pending queue as described in (a) above. This means that normal asynchronous event routines are called in an extension of normal (non-interrupt) processing between interrupt return and the main program. The routine is, therefore, not subject to the restrictions imposed on interrupt path routines.

c. In the Interrupt Path - Express Asynchronous Event

The event routine is called immediately, in the interrupt path. The routine must be at a 'near address' (see KL INIT EVENT). Under no circumstances may the routine enable interrupts.

KL EVENT enables interrupts unless it is called from the interrupt path.

Related entries:

KL INIT EVENT
KL NEXT SYNC
KL POLL SYNCHRONOUS
KL SYNC RESET

167: KL SYNC RESET #BCF5

Clear synchronous event queue.

Action:

The synchronous event queue is set empty - any outstanding events are simply discarded. The current event priority, used by KL POLL SYNCHRONOUS and KL NEXT SYNC to mask out lower priority events, is reset.

Entry conditions:

No conditions.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

It is the user's responsibility to ensure that the discarded events and any currently active events are reset. The event count of discarded events will be greater than zero, so any further 'kicks' will simply increment the count, but not add the event to the synchronous event queue - the events are, therefore, effectively disarmed.

Related entries:

KL DEL SYNCHRONOUS
KL NEXT SYNC
KL POLL SYNCHRONOUS

168: KL DEL SYNCHRONOUS #BCF8

Remove a synchronous event from the event queue.

Action:

The event is disarmed. If it is on the synchronous event queue then it is removed.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Deleting an event from the queue prevents the outstanding 'kicks' from being processed.

Before a synchronous event block is reset or reinitialised this routine should be used to ensure that it is not currently pending.

This routine enables interrupts.

Related entries:

KL DISARM EVENT
KL INIT EVENT
KL SYNC RESET

169: KL NEXT SYNC #BCFB

Get next event from the queue.

Action:

If there is an event on the synchronous event queue whose priority is greater than the current event priority (if any), then remove the event from the queue, set the current event priority to that of the event removed and return the previous event priority.

Entry conditions:

No conditions.

Exit conditions:

If there is an event to be processed:

Carry true.
HL contains the address of the event block.
A contains the previous event priority (if any).

If there is no event to be processed:

Carry false.
A and HL corrupt.

Always:

DE corrupt.
All other registers preserved.

Notes:

KL NEXT SYNC returns the address of the next event to be processed, if any, which it has taken off the synchronous event queue and whose priority has now been set as the event priority mask.

The procedure for processing synchronous events is as follows:

```
TRY_AGAIN:
    CALL   KL_NEXT_SYNC    ; return next event, if any
    JR     NC,??????       ; jump if no event to process
;
    PUSH   HL              ; save address of event
    PUSH   AF              ; save previous event priority
    CALL   KL_DO_SYNC      ; call the event routine
    POP    AF
    POP    HL
;
    CALL   KL_DONE_SYNC    ; reset the event priority mask, deal with
                           ; the event count and put the event back on
                           ; the queue if the count is still greater
                           ; than zero
    JR   TRY_AGAIN         ; see if any invents are still awaiting processing
```

The foreground program should call KL POLL SYNCHRONOUS regularly to check for outstanding events. KL POLL SYNCHRONOUS is a short routine in RAM, so calling it imposes little overhead. If there is an event outstanding then the above procedure should be invoked, and should be repeated until the event queue is empty.

The current event priority mechanism allows event routines to poll for, and process, events of higher priority. The priority returned by this routine must be preserved until it is passed to KL DONE SYNC.

KL NEXT SYNC enables interrupts.

Related entries:

KL DONE SYNC
KL DO SYNC
KL EVENT
KL INIT EVENT
KL POLL SYNCHRONOUS

170: KL DO SYNC #BCFE

Perform an event routine.

Action:

Call the event routine for a given event.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine is intended to be called to process an event after KL NEXT SYNC has found it to be pending. Use of this entry at any other time is not recommended.

See KL NEXT SYNC above for the general scheme for processing synchronous events.

KL DO SYNC does not itself affect the event count.

Related entries:

KL DONE SYNC
KL NEXT SYNC

171: KL DONE SYNC #BD01

Finish processing an event.

Action:

Once a synchronous event has been processed, by invoking its event routine via KL DO SYNC, this entry must be called to restore the current event priority and to deal with the event count. If the count remains greater than zero the event block is placed back on the synchronous event queue.

Entry conditions:

A contains the previous event priority.
HL contains the address of the event block.

Exit conditions:

AF, BC, DE and HL corrupt
All other registers preserved.

Notes:

This routine is intended to be called after calling KL NEXT SYNC, to find a pending event, and KL DO SYNC, to run the event routine. It uses the previous event priority and the event block address returned by KL NEXT SYNC. Other uses of this entry are not recommended.

See KL NEXT SYNC above for the general scheme for processing synchronous events.

Restoring the current event priority is an essential step in maintaining the synchronous event priority scheme.

If the event count is greater than zero then it is decremented. If the count is still greater than zero then there are further events outstanding and the event is placed back on the synchronous event queue. The event may be disarmed between KL NEXT SYNC and KL DONE SYNC. Setting the event count to one before calling KL DONE SYNC forces multiple events to be treated as a single event.

KL DONE SYNC may enable interrupts.

Related entries:

KL DO SYNC
KL NEXT SYNC

172: KL EVENT DISABLE #BD04

Disable normal synchronous events.

Action:

Prevent normal synchronous events from being processed but allow express synchronous events to be processed. This is achieved by setting the current event priority higher than any possible normal synchronous event priority.

Entry conditions:

No conditions.

Exit conditions:

HL corrupt.
All other registers preserved.

Notes:

KL EVENT DISABLE does not prevent events from being kicked. The effect is to 'mask off' all pending normal synchronous events so that they are hidden from the foreground program (when KL POLL SYNCHRONOUS or KL NEXT SYNC are called) and hence are not processed.

KL EVENT ENABLE reverses the effect of KL EVENT DISABLE.

It is not possible to disable synchronous events permanently from inside a synchronous event routine as the previous current event priority is restored when the event routine returns.

Related entries:

KL DISARM EVENT
KL EVENT ENABLE
KL NEXT SYNC
KL POLL SYNCHRONOUS

172: KL EVENT ENABLE #BD07

Enable normal synchronous events.

Action:

Allows normal and express synchronous events to be processed.

Entry conditions:

No conditions.

Exit conditions:

HL corrupt.
All other registers preserved.

Notes:

Events are enabled by default. KL EVENT ENABLE reverses the effect of KL EVENT DISABLE.

It is not possible to enable synchronous events permanently from inside a synchronous event routine as the current event priority which is used to disable events is restored when the event routine returns.

Related entries:

KL EVENT DISABLE
KL NEXT SYNC
KL POLL SYNCHRONOUS

174: KL DISARM EVENT #BD0A

Prevent an event from occurring.

Action:

Disarms the event by setting the event count to a negative value. Any further 'kicks' (calls of KL EVENT) for the event will be ignored, any outstanding events are discarded.

Entry conditions:

HL contains the address of the event block.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

KL DISARM EVENT should only be used with asynchronous events. Synchronous events may be disarmed by calling KL DEL SYNCHRONOUS, which also ensures that the event is not on the synchronous event queue.

The event may be re-armed by re-initialising it (KL INIT EVENT) or by setting the event count (byte 2 of the event block) to zero.

Related entries:

KL DEL SYNCHRONOUS
KL INIT EVENT

175: KL TIME PLEASE #BD0D

Ask the elapsed time.

Action:

The Kernel maintains a count which it increments on each time interrupt. The count, therefore, measures time in 1/300th of a second units. This routine returns the current count.

Entry conditions:

No conditions.

Exit conditions:

DEHL contains the four byte count (D contains the most significant byte and L the least significant byte).

All other registers preserved.

Notes:

The count is zeroised when the machine is turned on or reset. The count may be set to another starting value by KL TIME SET.

The count is not kept up to date if interrupts are disabled for long periods, such as while reading and writing the cassette.

The four byte count overflows after approximately:

	14,316,558	Seconds
=	238,609	Minutes
=	3,977	Hours
=	166	Days

This routine enables interrupts.

Related entries:

KL TIME SET

176: KL TIME SET #BD10

Set the elapsed time.

Action:

The Kernel maintains a count which it increments on each time interrupt. The count, therefore, measures time in 1/300th of a second units. This routine sets the count to a given value.

Entry conditions:

DEHL contains the four byte count to set (D contains the most significant byte and L the least significant byte).

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The four byte count overflows after approximately:

```
  14,316,558 Seconds
=    238,609 Minutes
=      3,977 Hours
=        166 Days
```

KL TIME SET may be used to set the count to the actual time of day, so that the Kernel then maintains a real clock rather than a simple measure of the time elapsed since the last reset.

The count is not kept up to date if interrupts are disabled for long periods, such as while reading and writing the cassette.

This routine enables interrupts.

Related entries:

KL TIME PLEASE

177: MC BOOT PROGRAM #BD13

Load and run a program.

Action:

Shut down as much of the system as possible then load a program into RAM and run it. If the load fails then the previous foreground program is restarted.

Entry conditions:

HL contains the address of the routine to call to load the program.

Exit conditions:

Does not exit!

Notes:

The system is partially reset before attempting to load the program. External interrupts are disabled, as are all timer, frame flyback and keyboard break events. Sound generation is turned off, indirections are set to their default routines and the stack is reset to the default system stack. This process ensures that no memory outside the firmware variables area is in use when loading the program. Overwriting an active event block or indirection routine could otherwise have unfortunate consequences.

The partial system reset does not change the ROM state or ROM selection. The routine run to load the program must be in accessible RAM or an enabled ROM. Note that the firmware jumpblock normally enables the lower ROM and disables the upper ROM and so the routine must normally be in RAM above #4000 or in the lower ROM.

The routine run to load the program is free to use any store from #0040 up to the base of the firmware variables area (#B100) and may alter indirections and arm external device interrupts as required. It should obey the following exit conditions:

> If the program loaded successfully:
>
> > Carry true.
> > HL contains the program entry point.
>
> If the program failed to load:
>
> > Carry false.
> > HL corrupt.
>
> Always:
>
> > A, BC, DE, IX, IY and other flags corrupt.

After a successful load the firmware is completely initialised (as at EMS) and the program is entered at the entry address returned by the load routine. Returning from the program will reset the system (perform RST 0).

After an unsuccessful load an appropriate error message is printed and the previous foreground program is restarted. If the previous foreground was itself a RAM program then the default ROM is entered instead as the program may have been corrupted during the failed loading.

Related entries:

CAS IN DIRECT
KL CHOKE OFF
MC START PROGRAM

178: MC START PROGRAM #BD16

Run a foreground program.

Action:

Fully initialise the system and enter a program.

Entry conditions:

HL contains the entry point address.
C contains the required ROM selection.

Exit conditions:

Never exits!

Notes:

HL and C comprise the 'far address' of the entry point of the foreground program (see section 2).

When entering a foreground program in ROM the ROM selection should be that required to select the appropriate ROM. When entering a foreground program in RAM the ROM selection should be used to enable or disable ROMs as the RAM program requires (ROM select addresses #FC..#FF).

This routine carries out a full EMS initialisation of the firmware before entering the program. Returning from the program will reset the system (perform RST 0).

MC START PROGRAM is intended for running programs in ROM or programs that have already been loaded into RAM. To load and run a RAM program use MC BOOT PROGRAM.

Related entries:

MC BOOT PROGRAM
RESET ENTRY (RST0)

179: MC WAIT FLYBACK #BD19

Wait for frame flyback.

Action:

Wait until frame flyback occurs.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

Frame flyback is a signal generated by the CRT controller to signal the start of the vertical retrace period. During this period the screen is not being written and so major operations can be performed on the screen without producing unsightly effects. A prime example is rolling the screen.

The frame flyback signal only lasts for a couple of hundred microseconds but the vertical retrace period is much longer than this. However, there will be a ticker interrupt in the middle of frame flyback which may cause the foreground processing to be suspended for a significant length of time. It is important, therefore, to perform any critical processing as soon after frame flyback is detected as is possible.

Related entries:

KL ADD FRAME FLY

180: MC SET MODE #BD1C

Set the screen mode.

Action:

Load the hardware with the required screen mode.

Entry conditions:

A contains the required mode.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The required mode is checked and no action is taken if it is invalid. If it is valid then the new value is sent to the hardware.

The screen modes are:

0:	160 x 200 pixels,	20 x 25 characters.
1:	320 x 200 pixels,	40 x 25 characters.
2:	640 x 200 pixels,	80 x 25 characters.

Altering the screen mode without notifying the Screen Pack will produce peculiar effects on the screen. In general SCR SET MODE should be called to change screen mode. This, in its turn, sets the new mode into the hardware.

Related entries:

SCR SET MODE

181: MC SCREEN OFFSET #BD1F

Set the screen offset.

Action:

Load the hardware with the offset of the first byte on the screen inside a 2K screen block and which 16K block the screen memory is located in.

Entry conditions:

A contains the new screen base.
HL contains the new screen offset.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The screen base address is masked with #C0 to make sure it refers to a valid 16K memory area. The default screen base is #C0 (the screen is underneath the upper ROM).

The screen offset is masked with #07FE to make it legal. Note that bit 0 is ignored as the hardware only uses even offsets.

If the screen base or offset is changed without notifying the Screen Pack then unexpected effects may occur on the screen. In general SCR SET BASE or SCR SET OFFSET should be called. These, in their turn, send the values to the hardware.

Related entries:

SCR SET BASE
SCR SET OFFSET

182: MC CLEAR INKS #BD22

Set all inks to one colour.

Action:

Set the colour of the border and set the colour of all the inks. All the inks are set to the same colour thus giving the impression that the screen has been cleared instantly.

Entry conditions:

DE contains the address of an ink vector.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The ink vector has the form:

Byte 0:	Colour for the border.
Byte 1:	Colour for all inks.

The colours supplied are the numbers used by the hardware rather than the grey scale numbers supplied to SCR SET INK (see Appendix V).

After the screen has been cleared (or whatever) the correct ink colours can be set by calling MC SET INKS.

This routine sets the colours for all 16 inks whether they can be displayed on the screen in the current mode or not.

This ink clearing technique is used by the Screen Pack when clearing the screen or changing mode (by SCR CLEAR and SCR SET MODE).

Related entries:

MC SET INKS

183: MC SET INKS #BD25

Set colours of all the inks.

Action:

Set the colours of all the inks and the border.

Entry conditions:

DE contains the address of an ink vector.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The ink vector passed has the following layout:

Byte 0:	Colour for the border.
Byte 1:	Colour for ink 0.
Byte 2:	Colour for ink 1.
..	..
Byte 16:	Colour for ink 15.

The colours supplied are the numbers required by the hardware rather than the grey scale numbers supplied to SCR SET INK (see Appendix V).

This routine sets the colours for all inks including those that cannot be visible in the current screen mode. However, it is only necessary to supply sensible colours for the visible inks.

The Screen Pack sets the colours for all the inks each time the inks flash and after an ink colour has been changed (by calling SCR SET INK or SCR SET BORDER).

Related entries:

MC CLEAR INKS
SCR SET BORDER
SCR SET INK

184: MC RESET PRINTER #BD28

Reset the printer indirection.

Action:

Set the printer indirection, MC WAIT PRINTER, to its default routine.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

MC WAIT PRINTER
MC PRINT CHAR

185: MC PRINT CHAR #BD2B

Try to send a character to the Centronics port.

Action:

Send a character to the printer (Centronics port) or time out if the printer is busy for too long.

Entry conditions:

A contains the character to send (bit 7 ignored).

Exit conditions:

If the character was sent OK:

Carry true.

If the printer timed out:

Carry false.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

This routine calls the Machine Pack indirection MC WAIT PRINTER to send the character. The default indirection routine waits for the Centronics port to become non-busy then sends the character. If the port remains busy for too long (approximately 0.4 seconds) then the routine times out and the character is not sent. This time out is provided so that the caller can test for break whilst driving the printer.

Related entries:

MC RESET PRINTER
MC WAIT PRINTER

186: MC BUSY PRINTER #BD2E

Test if the Centronics port is busy.

Action:

Test if the printer (Centronics port) is busy.

Entry conditions:

No conditions.

Exit conditions:

If Centronics port is busy:

Carry true.

If Centronics port is idle:

Carry false.

Always:

Other flags corrupt.
All other registers preserved.

Notes:

This routine has no other effects.

Related entries:

MC SEND PRINTER

187: MC SEND PRINTER #BD31

Send a character to the Centronics port.

Action:

Send a character to the printer (Centronics port) which must not be busy.

Entry conditions:

A contains the character to send (bit 7 ignored).

Exit conditions:

Carry true.
A and other flags corrupt.
All other registers preserved.

Notes:

The printer must not be busy when a character is sent. The higher level routine MC PRINT CHAR will automatically wait for the printer to become non-busy and should be used in preference.

Related entries:

MC BUSY PRINTER
MC PRINT CHAR

188: MC SOUND REGISTER #BD34

Send data to a sound chip register.

Action:

Set a sound chip sound register. This is a rather convoluted action because of the way the hardware has been designed.

Entry conditions:

A contains the sound chip register number.
C contains the data to send.

Exit conditions:

AF and BC corrupt.
All other registers preserved.

Notes:

This routine enables interrupts.

Related entries:

None!

189: JUMP RESTORE #BD37

Restore the standard jumpblock.

Action:

Set the main firmware jumpblock to its standard state as described in sections 13.1 and 14.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This routine may be used to restore the jumpblock to its standard routines after the user has changed entries in it. The whole of the jumpblock is set up so care must be taken if other programs may have patched it.

The indirections jumpblock is set up piecemeal by the various packs' initialisation and reset routines. JUMP RESTORE does not set up the indirections.

Related entries:

GRA RESET
KM RESET
MC RESET PRINTER
SCR RESET
TXT RESET

15 The Firmware Indirections.

This section gives the detailed entry and exit conditions and effects of the routines in the indirections jumpblock. See section 13.2 for a list of these routines.

The firmware indirections are taken by the firmware at key points. They allow the user to intercept and alter a number of firmware actions without having to provide a complete new firmware package.

The descriptions given are for the default settings of the indirections. Replacement routines need not perform all the actions that the default routine performs although they are advised to do so.

IND: TXT DRAW CURSOR #BDCD

Place the cursor blob on the screen (if enabled).

Default action:

If the cursor is enabled and turned on the cursor blob is drawn on the screen. If not then no action is taken. The current text position is forced into the window (see TXT VALIDATE) and the cursor blob is written at the resulting position. The cursor blob is an inverse patch. This routine will only be called twice if TXT UNDRAW CURSOR is called in between.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

This indirection is provided to allow the user to change the form of the cursor blob. See TXT PLACE CURSOR for a description of how the cursor blob is normally written.

The Text VDU routines call this indirection whenever the cursor is placed on the screen. All the Text VDU routines that read from the screen, write to the screen or change the current position remove the cursor (using TXT UNDRAW CURSOR) before performing their action and place it back on the screen afterwards (using TXT DRAW CURSOR). An example of such a routine is TXT WR CHAR that writes a character on the screen.

This indirection is set up when TXT INITIALISE or TXT RESET is called.

Related entries:

TXT PLACE CURSOR
TXT UNDRAW CURSOR

IND: TXT UNDRAW CURSOR #BDD0

Remove the cursor blob from the screen (if enabled).

Default action:

If the cursor is enabled and turned on the cursor blob is removed from the screen. If not then no action is taken. The cursor blob is an inverse patch. This routine will only be called after TXT DRAW CURSOR has been used to place the cursor on the screen.

Entry conditions:

No conditions.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

This indirection is provided to allow the user to change the form of the cursor blob. See TXT REMOVE CURSOR for a description of how the cursor blob is normally removed.

The Text VDU routines call this indirection to remove the cursor from the screen. All the Text VDU routines that read from the screen, write to the screen or change the current position remove the cursor (using TXT UNDRAW CURSOR) before performing their action and place it back on the screen afterwards (using TXT DRAW CURSOR). An example of such a routine is TXT WR CHAR that writes a character on the screen.

This indirection is set up when TXT INITIALISE or TXT RESET is called.

Related entries:

TXT DRAW CURSOR
TXT REMOVE CURSOR

IND: TXT WRITE CHAR #BDD3

Write a character onto the screen.

Default action:

Put a character on the screen at a character position.

Entry conditions:

A contains the character to write.
H contains the physical column to write at.
L contains the physical row to write at.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The character position to write at is given in physical coordinates. i.e. Row 0, column 0 is the top left corner of the screen. The position is not checked for legality.

TXT WRITE CHAR is called by TXT WR CHAR to print a character on the screen. The removing of the cursor blob and the calculation of the new current position are performed by TXT WR CHAR and not by TXT WRITE CHAR.

This indirection is set up when TXT INITIALISE or TXT RESET is called.

Related entries:

TXT OUTPUT
TXT UNWRITE
TXT WR CHAR

IND: TXT UNWRITE #BDD6

Read a character from the screen.

Default action:

Try to read a character from the screen at a character position.

Entry conditions:

H contains the physical column to read from.
L contains the physical row to read from.

Exit conditions:

If a readable character was found:

    Carry true.
    A contains the character read.

If no recognisable character was found:

    Carry false.
    A contains zero.

Always:

    BC, DE, HL and other flags corrupt.
    All other registers preserved.

Notes:

The character position to read from is given in physical coordinates, i.e. Row 0, column 0 is the top left corner of the screen. The position is not checked for legality.

This indirection is called by TXT RD CHAR to read a character from the screen. TXT RD CHAR removes the cursor from the screen before calling this indirection.

The read is performed by comparing the matrix found on the screen with the matrices used to generate characters. As a result changing a character matrix, changing the pen or paper inks or changing the screen (e.g. drawing a line through a character) may make the character unreadable. In particular the cursor blob will cause confusion and so it should not be on the screen.

Special precautions are taken against space being generated. Initially the character is read assuming that the character was written in the current pen ink. If this fails to generate a recognisable character or it generates space (character #20) then another try is made by assuming that the background to the character was written in the current paper ink.

The characters are scanned starting with #00 and finishing with #FF. Thus, if there are two possible character matrices that match the screen, the smaller of the two characters will be returned.

This indirection is set up when TXT INITIALISE or TXT RESET is called.

Related entries:

TXT RD CHAR
TXT WRITE CHAR

IND: TXT OUT ACTION #BDD9

Output a character or control code.

Default action:

Print a character on the screen or obey a control code (characters #00..#1F). Works on the currently selected stream (except as noted below).

Entry conditions:

A contains the character or code.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

This indirection is called by TXT OUTPUT to do the work of printing characters or obeying the control codes. It is provided to allow the user to change the method of dealing with characters and control codes or to allow the user to redirect output (to the printer for example). TXT OUTPUT merely preserves the registers around the call of TXT OUT ACTION.

Control codes may take up to 9 parameters. These are the characters sent following the initial control code. The characters sent are stored in a buffer until sufficient have been received to make up all the required parameters. The control code buffer is only long enough to accept 9 parameter characters.

There is only one control code buffer which is shared between all the streams. It is, therefore, possible to get unpredictable results if the output stream is changed part of the way through sending a control code sequence.

If the VDU is disabled then no characters will be printed on the screen. However, control codes will still be obeyed.

If the graphics character write mode is enabled then all characters and control codes are printed using the Graphics VDU (see GRA WR CHAR) and are not obeyed. Normally characters are written by the Text VDU (see TXT WR CHAR).

This indirection is set up when TXT INITIALISE or TXT RESET is called.

Related entries:

TXT OUTPUT
TXT WR CHAR

IND: GRA PLOT #BDDC

Plot a point.

Default action:

Check if the point lies inside the current window and if so write it in the current graphics pen ink and using the current graphics write mode. The current graphics position is always moved to the specified point.

Entry conditions:

DE contains the user X coordinate of the point to plot.
HL contains the user Y coordinate of the point to plot.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The position of the point to plot is given in user coordinates, i.e. relative to the user origin.

This indirection is called by GRA PLOT RELATIVE and GRA PLOT ABSOLUTE to plot the point requested. It is provided to allow the user to change the method for plotting (to output to an X-Y plotter for example). GRA PLOT RELATIVE converts from relative to user coordinates and then calls this indirection; GRA PLOT ABSOLUTE calls this indirection immediately.

To write the point on the screen the SCR WRITE indirection is used. Thus the point is plotted using the current graphics write mode.

This indirection is set up when GRA INITIALISE or GRA RESET is called.

Related entries:

GRA PLOT ABSOLUTE
GRA PLOT RELATIVE
GRA TEST
SCR WRITE

IND: GRA TEST #BDDF

Test a point.

Default action:

Check if the point is inside the graphics window and return the ink it is currently set to if so. Otherwise, return the current graphic paper ink. The current graphics position is always moved to the specified point.

Entry conditions:

DE contains the user X coordinate of the point to test.
HL contains the user Y coordinate of the point to test.

Exit conditions:

A contains the decoded ink of the specified point.

BC, DE, HL and flags corrupt.
All other registers preserved.

Notes:

The position of the point to test is given in user coordinates, i.e. relative to the user origin.

This indirection is used by GRA TEST RELATIVE and GRA TEST ABSOLUTE to test the point requested. It is provided to allow the user to change the method for testing (comparing with the current pen ink for example). GRA TEST RELATIVE converts from relative to user coordinates and then calls this indirection; GRA TEST ABSOLUTE calls this indirection immediately.

To test the ink of a point inside the window the SCR READ indirection is used.

This indirection is set up when GRA INITIALISE or GRA RESET is called.

Related entries:

GRA PLOT
GRA TEST ABSOLUTE
GRA TEST RELATIVE
SCR READ

IND: GRA LINE #BDE2

Draw a line.

Default action:

Draw a line in the current graphics pen ink between the current graphics position and the given endpoint using the current graphics write mode. Points on the line that lie outside the current graphics window will not be plotted. The current graphics position is always moved to the specified endpoint.

Entry conditions:

DE contains the user X coordinate of the endpoint.
HL contains the user Y coordinate of the endpoint.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

The position of the endpoint is given in user coordinates, i.e. relative to the user origin.

This indirection is used by GRA LINE RELATIVE and GRA LINE ABSOLUTE to draw the line requested. It is provided to allow the user to change the method for line drawing (to output to an X-Y plotter for example). GRA LINE RELATIVE converts from relative to user coordinates and then calls the indirection; GRA LINE ABSOLUTE calls the indirection immediately.

The line is split up into horizontal or vertical sections that are drawn separately (see SCR HORIZONTAL and SCR VERTICAL). The SCR WRITE indirection is called to write the points in these sections. This means that the line is plotted using the current graphics write mode.

This indirection is set up when GRA INITIALISE or GRA RESET is called.

Related entries:

GRA LINE ABSOLUTE
GRA LINE RELATIVE
SCR HORIZONTAL
SCR VERTICAL

IND: SCR READ #BDE5

Read a pixel from the screen.

Default action:

Read a pixel from the screen and decode its ink.

Entry conditions:

HL contains the screen address of the pixel.
C contains the mask for the pixel.

Exit conditions:

A contains the decoded ink that the pixel was set to.

Flags corrupt.
All other registers preserved.

Notes:

The mask supplied must be a mask for a single pixel otherwise the decoding of the ink read from the screen will not work correctly.

This indirection is set up when SCR INITIALISE or SCR RESET is called. It is called by GRA TEST.

Related entries:

GRA TEST
SCR WRITE

IND: SCR WRITE #BDE8

Write pixel(s) using the current graphics write mode.

Default action:

Plot a pixel or pixels on the screen using the current graphics write mode.

Entry conditions:

HL contains the screen address of the pixel(s).
C contains the mask for the pixel(s).
B contains the encoded ink to plot with.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The pixel mask supplied can be for a single pixel or more than one pixel (or even no pixels). The ink supplied should be encoded to cover the whole of a byte (see SCR INK ENCODE).

The pixel is plotted using the current Graphics VDU write mode. These modes are:

FORCE	Pixel is set to the new ink irrespective of the old ink.
XOR	Pixel is set to the ink formed by exclusive-oring the new ink for the pixel and its current setting.
AND	Pixel is set to the ink formed by anding the new ink for the pixel and its current setting.
OR	pixel is set to the ink formed by oring the new ink for the pixel and its current setting.

The write mode can be set by calling SCR ACCESS appropriately.

This indirection is called by all Graphics VDU write routines, in particular GRA PLOT RELATIVE, GRA PLOT ABSOLUTE, GRA LINE RELATIVE, GRA LINE ABSOLUTE and GRA WR CHAR, to plot pixels on the screen. It is provided to allow the user to intercept the lowest level of point plotting (perhaps to add yet another plotting mode).

This indirection is set up when SCR INITIALISE or SCR RESET is called.

Related entries:

GRA PLOT
SCR ACCESS
SCR PIXELS
SCR READ

IND: SCR MODE CLEAR #BDEB

Clear the screen to ink 0.

Default action:

Clear the screen memory to zeros. This indirection is provided to allow the user to prevent the screen being cleared after the mode is changed.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

Normally this indirection performs the actions described in SCR CLEAR.

This indirection is set up when SCR INITIALISE or SCR RESET is called.

N.B. When this indirection is called the text and graphics VDUs are in non-standard states.

Related entries:

SCR CLEAR
SCR SET MODE

IND: KM TEST BREAK #BDEE

Test for break (or reset).

Default action:

Test if the escape key is pressed, if not then no action is taken. If escape, shift and control are all pressed and no other keys then the system is reset. Otherwise, a break event is reported (see KM BREAK EVENT).

Entry conditions:

Interrupts disabled.
C contains shift and control key states.

Exit conditions:

AF and HL corrupt.
All other registers preserved.

Notes:

This indirection is called by the firmware from the interrupt path. Thus interrupts are disabled and they must remain disabled.

If bit 7 of C is set then the control key is pressed. If bit 5 of C is set then one of the shift keys is pressed.

This indirection is called after the keys have been scanned and the escape key was found to have been pressed. It is provided to allow the user to alter the action of a break (particularly to prevent the system reset, see RESET ENTRY).

This indirection is set up when KM INITIALISE or KM RESET is called.

Related entries:

KM BREAK EVENT

IND: MC WAIT PRINTER #BDF1

Print a character or time out.

Default action:

Wait for the Centronics port to become not busy and then send a character to it. If the port remains busy for a long time the routine times out and the character is not sent.

Entry conditions:

A contains the character to send.

Exit conditions:

If the character was sent OK:

Carry true.

If the Centronics port timed out:

Carry false.

Always:

A and BC corrupt.
All other registers preserved.

Notes:

This indirection is provided to allow the user to drive the printer in a different way. For example, 'escape sequences' could be handled or the time out could be changed.

This indirection is called by the routine MC PRINT CHAR. It tests whether the printer is busy in the same way as MC BUSY PRINTER and sends the character in the same way as MC SEND PRINTER.

This indirection is set up when MC RESET PRINTER is called.

Related entries:

MC BUSY PRINTER
MC PRINT CHAR
MC SEND PRINTER

16 The High Kernel Jumpblock.

Separate from the main firmware jumpblock is a small jumpblock for Kernel routines associated with ROM state and ROM selection. The routines accessed through this jumpblock are all RAM resident, to avoid confusion while the ROM state and ROM select are changed! The RAM area is copied out of ROM during the power-up initialisation. The jumpblock should not be altered by the user.

The entry KL POLL SYNCHRONOUS is the 'odd man out' amongst the routines in this jumpblock. Unlike the other synchronous event handling routines, which are in the lower ROM, this routine is RAM resident. This minimises the overhead involved in polling for synchronous events.

A brief listing of the entries in this jumpblock can be found in section 13.3. A discussion of ROMs and the memory map can be found in section 2, further discussion of ROMs can be found in section 9 and a discussion of events can be found in section 11.

HI: KL U ROM ENABLE #B900

Enable the upper ROM.

Action:

Enables the currently selected upper ROM. Reading from addresses #C000 and up will now return the contents of the ROM.

Entry conditions:

No conditions.

Exit conditions:

A contains the previous ROM state.

Flags corrupt.
All other registers preserved.

Notes:

The mechanisms provided for calling subroutines in the upper ROM and for selecting upper ROMs automatically enable the ROM as required. This routine is used by the firmware but is otherwise of little use.

The previous ROM state may be passed to KL ROM RESTORE to reset the state to what it was before this routine was called.

This routine enables interrupts.

Related entries:

KL L ROM ENABLE
KL ROM RESTORE
KL ROM SELECT
KL U ROM DISABLE

HI: KL U ROM DISABLE #B903

Disable the upper ROM.

Action:

Disables the upper ROM. Reading from addresses #C000 and up will now return the contents of the RAM.

Entry conditions:

No conditions.

Exit conditions:

A contains the previous ROM state.

Flags corrupt.
All other registers preserved.

Notes:

Disabling the upper ROM gives read access to the top 16K of RAM, which is usually used as the screen memory. Note that the mapping of a location in screen memory to pixels on the screen depends on the mode and on the screen offset.

It is inadvisable to disable the upper ROM while executing instructions in it!

The previous ROM state may be passed to KL ROM RESTORE to reset the state to what it was before this routine was called.

This routine enables interrupts.

Related entries:

KL L ROM DISABLE
KL ROM RESTORE
KL U ROM ENABLE

HI: KL L ROM ENABLE #B906

Enable the lower ROM.

Action:

Enables the lower ROM. Reading from addresses below #4000 will now return the contents of the ROM.

Entry conditions:

No conditions.

Exit conditions:

A contains the previous ROM state.

Flags corrupt.
All other registers preserved.

Notes:

In general the lower ROM is disabled except when a firmware routine is called. The firmware jumpblock arranges to enable the lower ROM automatically and to disable it again when the routine returns. This routine is used by the firmware but is otherwise of little use.

The previous ROM state may be passed to KL ROM RESTORE to reset the state to what it was before this routine was called.

This routine enables interrupts.

Related entries:

KL L ROM DISABLE
KL ROM RESTORE
KL U ROM ENABLE

111: KL L ROM DISABLE #B909

Disable the lower ROM.

Action:

Disables the lower ROM. Reading from addresses below #4000 will now return the contents of the RAM.

Entry conditions:

No conditions.

Exit conditions:

A contains the previous ROM state.

Flags corrupt.
All other registers preserved.

Notes:

In general the lower ROM is disabled except when a firmware routine is called. The firmware jumpblock arranges to enable the lower ROM automatically and to disable it again when the routine returns.

The previous ROM state may be passed to KL ROM RESTORE to reset the state to what it was before this routine was called.

This routine enables interrupts.

Related entries:

KL L ROM ENABLE
KL ROM RESTORE
KL U ROM DISABLE

HI: KL ROM RESTORE #B90C

Restore the previous ROM state.

Action:

The ROM state change routines all return a value giving the previous ROM state. Given that value KL ROM RESTORE will reset the state to what it was before the change.

Entry conditions:

A contains the previous ROM state.

Exit conditions:

AF corrupt.
All other registers preserved.

Notes:

The previous ROM state is the value returned by one of:

KL U ROM ENABLE
KL U ROM DISABLE
KL L ROM ENABLE
KL L ROM DISABLE
KL ROM SELECT

It is possible to use KL U ROM DISABLE to reverse the effect of a call of KL U ROM ENABLE (amongst various other combinations). However, calling KL ROM RESTORE is the prefered method since it restores the state to what it was, which might have been enabled anyway.

This routine enables interrupts.

Related entries:

KL L ROM DISABLE
KL L ROM ENABLE
KL ROM SELECT
KL U ROM DISABLE
KL U ROM ENABLE

HI: KL ROM SELECT #B90F

Select a particular upper ROM.

Action:

Select a given upper ROM and enable the upper ROM.

Entry conditions:

C contains the ROM select address of the required ROM.

Exit conditions:

C contains the ROM select address of the previously selected ROM.
B contains the previous ROM state.

AF corrupt.
All other registers preserved.

Notes:

The previous state can be passed to KL ROM RESTORE to reset the ROM enable to what it was. Both the previous state and the previous selection can be passed to KL ROM DESELECT to restore the state to what it was and to select the previously selected ROM again.

The mechanisms provided for calling routines in expansion ROMs automatically perform ROM selection as required (see section 2).

It is inadvisable to select another upper ROM whilst executing instructions in the upper ROM.

This routine enables interrupts.

Related entries:

KL CURR SELECTION
KL PROBE ROM
KL ROM DESELECT
KL ROM RESTORE

HI: KL CURR SELECTION #B912

Ask which upper ROM is currently selected.

Action:

Returns the ROM select address of the currently selected upper ROM.

Entry conditions:

No conditions.

Exit conditions:

A contains the ROM select address of the currently selected ROM.

All other registers and flags preserved.

Notes:

It is not possible to predict the ROM select address at which any particular expansion ROM will be fitted. The 'far address' used to reference subroutines in expansion ROMs includes a ROM select byte which must be set up at run time. This routine returns the ROM select address of the current ROM so that it can set up suitable 'far addresses'.

Related entries:

KL PROBE ROM
KL ROM SELECT

HI: KL PROBE ROM #B915

Ask class and version of a ROM.

Action:

The first few bytes of all upper ROMs contain information in a standard form about the ROM. This routine extracts the class, mark number and version number bytes from the ROM at the given ROM select address.

Entry conditions:

C contains the ROM select address of the ROM to probe.

Exit conditions:

A contains the ROM's class.
L contains the ROM's mark number.
H contains the ROM's version number.

B and flags corrupt.
All other registers preserved.

Notes:

The ROM class returned may take any of the following values:

0:	Foreground ROM.
1:	Background ROM.
2:	Extension foreground ROM.
#80:	On board ROM (the built in BASIC foreground program).

Selecting a ROM address where no ROM is fitted implicitly selects the on-board ROM and so it will return #80 as its class.

The meaning of the mark and version numbers depends on the ROM.

See section 9 for a description of expansion ROMs.

This routine enables interrupts.

Related entries:

KL ROM SELECT
KL CURR SELECTION

HI: KL ROM DESELECT #B918

Restore previous upper ROM selection.

Action:

Set the ROM state and upper ROM selection to what they were before KL ROM SELECT was called.

Entry conditions:

C contains the ROM select address of the previously selected ROM.
B contains the previous ROM state.

Exit conditions:

C contains the ROM select address of the currently selected ROM.

B corrupt.
All other registers and flags preserved.

Notes:

The previous ROM selection and state are the values returned by KL ROM SELECT. The currently selected ROM returned by this routine is the ROM that was selected by calling KL ROM SELECT (unless further selections have been made).

The mechanisms provided for calling subroutines in expansion ROMs automatically perform ROM selection as required.

It is inadvisable to select another upper ROM whilst executing instructions in the upper ROM.

This routine enables interrupts.

Related entries:

KL CURR SELECTION
KL ROM RESTORE
KL ROM SELECT

Move store (LDDR) with ROMs turned off.

Action:

Performs an LDDR instruction (LoaD Decrement and Repeat) with both upper and lower ROMs disabled.

Entry conditions:

BC, DE, HL as required by LDDR instruction.

Exit conditions:

F, BC, DE, HL as set by LDDR instruction.
All other registers preserved.

Notes:

This routine may be used to move areas of RAM irrespective of the ROM state.

This routine enables interrupts.

Related entries:

KL LDIR
RAM LAM (RST4)

Move store (LDIR) with ROMs turned off.

Action:

Performs an LDIR instruction (LoaD Increment and Repeat) with both upper and lower ROMs disabled.

Entry conditions:

BC, DE, HL as required by the LDIR instruction.

Exit conditions:

F, BC, DE, HL as set by the LDIR instruction.
All other registers preserved.

Notes:

This routine may be used to move areas of RAM irrespective of the ROM state.

This routine enables interrupts.

Related entries:

KL LDDR
RAM LAM (RST4)

HI: KL POLL SYNCHRONOUS #B921

Check if an event with higher priority than the current event is pending.

Action:

If the synchronous event queue is not empty then the priority of the highest priority pending event is compared with the current event's priority (if any).

Entry conditions:

No conditions.

Exit conditions:

If there is a higher priority event pending:

Carry true.

If there is no higher priority event pending:

Carry false.

Always:

A and other flags corrupt.
All other registers preserved.

Notes:

This routine is in the high jumpblock to minimise the overhead of polling for synchronous events. If the synchronous event queue is empty the routine takes only a few instructions.

While a synchronous event is being processed the Kernel remembers its priority. The synchronous event routine may itself poll the synchronous event queue, but only events of a higher priority than itself are notified to it.

This routine may enable interrupts.

Related entries:

KL EVENT
KL DONE SYNC
KL DO SYNC
KL NEXT SYNC

17 The Low Kernel Jumpblock.

The bottom of memory, from #0000 to #003F inclusive, is occupied by the code for the restart (RST) instructions and a number of Kernel entries. Most of these entries are concerned with access to subroutines in ROM and RAM. The RST's are:

RST 0 performs a system reset.

RST instructions 1 to 5 inclusive have been used to extend the Z80 instruction set to provide extra CALL and JUMP instructions, which use addresses extended to include ROM state and ROM select components.

RST 6 is available to the user.

RST 7 is used by interrupts.

Since all the entries supplied must be available whether the lower ROM is enabled or not, the area is copied into RAM from the ROM during power-up initialisation.

The user is not intended to alter this jumpblock (except where noted in the USER RESTART and EXT INTERRUPT areas). If the user does change the area then it is the user's responsibility to ensure that the changes do not affect other programs. To some extent this can be achieved by ensuring that the lower ROM is always enabled when other programs are running. However, since the other programs may disable the lower ROM this is insufficient in most cases. Ideally the original jumpblock contents should be restored where there is any doubt.

Section 2 contains a discussion of ROMs and the memory map and section 9 contains a general discussion of external ROMs. A brief list of the routines in this area can be found in section 13.4.

LOW: RESET ENTRY RST 0 #0000

Completely reset the machine as if powered up.

Action:

When the machine is first turned on execution starts here. Calling or jumping to #0000, or executing RST 0, resets the machine to its initial power-up state.

Entry conditions:

No conditions.

Exit conditions:

Does not return!

Notes:

All hardware is reset and the firmware is completely initialised. Once all tables and jumpblocks have been set up, control is passed to the default entry in ROM 0 (see section 9).

Related Entries.

MC START PROGRAM

LOW: LOW JUMP RST 1 #0008

Jump to lower ROM or RAM, takes inline 'low address' to jump to.

Action:

RST 1 is used to extend the instruction set. It is an expanded form of the jump instruction. It should be followed by a 2 byte 'low address' which specifies the location to jump to and the required ROM state.

Entry conditions:

All registers and flags are passed to the target routine untouched.

Exit conditions:

All registers and flags are as set by the target routine.

Notes:

The 'low address' following the restart instruction is laid out as follows:

Bit:	15	14	13 … 0
	U	L	Address

If the 'U' bit is set then the upper ROM is disabled.
If the 'L' bit is set then the lower ROM is disabled.
'Address' is the actual address of the target routine to jump to once the ROM state has been set.

When the target routine returns the ROM state is restored to what it was before the jump. To accomplish this 4 bytes are pushed onto the stack and so care should be taken when indexing up the stack (to find the address of inline parameters, for example).

The LOW JUMP, RST 1, 'instruction' may replace the first byte of a JP (jump) instruction. It is intended for use in jumpblocks. The main firmware jumpblock is made up almost exclusively of LOW JUMP 'instructions'.

It is assumed that the destination of the jump is a routine which will return in the usual way. The restart instruction itself does not return. The value at the top of the stack when a LOW JUMP is executed must, therefore, be a return address.

Executing a LOW JUMP enables interrupts.

LOW: KL LOW PCHL #000B

Jump to lower ROM or RAM.
Register HL contains the 'low address' to jump to.

Action:

Takes a 'low address' as a parameter and jumps to it. The 'low address' specifies both the address to jump to and the ROM state required.

Entry conditions:

HL contains the 'low address' to jump to.

All registers and flags are passed to the target routine untouched.

Exit conditions:

All registers and flags are as set by the target routine.

Notes:

The 'low address' is laid out as follows:

Bit:	15	14	13	0
	U	L	Address	

If the 'U' bit is set then the upper ROM is disabled.
If the 'L' bit is set then the lower ROM is disabled.
'Address' is the actual address to jump to, once the ROM state has been set.

When the target routine returns the ROM state is restored to what it was before the jump. To accomplish this 4 bytes are pushed onto the stack and so care should be taken when indexing up the stack (to find the address of inline parameters, for example).

It is assumed that the destination of the jump is a routine which will return in the usual way. The value at the top of the stack when a LOW PCHL is executed must, therefore, be a return address.

Interrupts are enabled.

Related entries:

KL FAR ICALL
KL FAR PCHL
LOW JUMP (RST1)
PCHL INSTRUCTION

LOW: PCBC INSTRUCTION #000E

Jump to address in BC.

Action:

Equivalent to the JP (HL) instruction (or PCHL in some assembler dialects), except that the destination is in BC not HL.

Entry conditions:

BC contains the address to jump to.

All registers and flags are passed to the target routine untouched.

Exit conditions:

All registers and flags are as set by the target routine.

Notes:

Calling PCBC INSTRUCTION is a useful way of invoking a routine whose address has been picked out of a table or otherwise established at run time.

Related entries:

KL FAR PCHL
KL LOW PCHL
KL SIDE PCHL
PCDE INSTRUCTION
PCHL INSTRUCTION

LOW: SIDE CALL RST 2 #0010

Call to a sideways ROM, takes inline 'side address' to call.

Action:

RST 2 is used to extend the instruction set. It is an expanded form of the CALL instruction. It should be followed by a 2 byte 'side address' which specifies the location to call and the required ROM selection.

Entry conditions:

All registers and flags are passed to the target routine untouched except for IY (which is set to point at a background ROM's upper data area).

Exit conditions:

IY preserved.
All other registers and flags are as set by the target routine.

Notes:

The 'side address' following the restart instruction is laid out as follows:

Bit:	15 14	13 ... 0
	Off	Address

> 'Off' gives a value in the range 0..3, which, when added to the ROM select address of the main foreground ROM, gives the ROM select address of the required ROM.
>
> After #C000 has been added to it, 'address' is the address of the routine to call.

The target routine returns to the instruction immediately following the inline 'side address'. The ROM select and ROM state are restored to what they were before the call. To accomplish this 6 bytes are pushed onto the stack and so care should be taken when indexing up the stack (to find the address of inline parameters, for example).

When the target routine is entered the lower ROM is disabled and the appropriate upper ROM is selected and enabled.

SIDE CALLs are provided to support foreground programs split over a number of ROMs (up to four). See section 9 on expansion ROMs.

Interrupts are enabled.

Related entries:

FAR CALL (RST3)
KL SIDE PCHL

LOW: KL SIDE PCHL #0013

Call to a sideways ROM, HL contains 'side address' to call.

Action:

Takes a 'side address' and calls it. The 'side address' specifies the address of the routine to call and which upper ROM to select.

Entry conditions:

HL contains the 'side address' to call.

All registers and flags are passed to the target routine untouched except for IY (which is set to point at a background ROM's upper data area).

Exit conditions:

IY preserved.
All other registers and flags are as set by the target routine.

Notes:

The 'side address' is laid out as follows:

Bit: 15 14	13 ... 0
Off	Address

> 'Off' gives a value in the range 0..3, which, when added to the ROM select address of the main foreground ROM, gives the ROM select address of the required ROM.
>
> After #C000 has been added to it, 'address' is the address of the routine to call.

When the target routine is entered the lower ROM is disabled and the appropriate upper ROM is selected and enabled.

When the target routine returns the ROM select and ROM state are restored to what they were before the call. This is accomplished by pushing 6 bytes onto the stack and so care should be taken when indexing up the stack (to find the address of inline parameters, for example).

Side calls are provided to support foreground programs split over a number of ROMs (up to four). See section 9 on external ROMs.

Interrupts are enabled.

Related entries:

FAR CALL (RST3)
KL FAR ICALL
KL FAR PCHL

LOW: PCDE INSTRUCTION #0016

Jump to address in DE.

Action:

Equivalent to the JP (HL) instruction (or PCHL in some assembler dialects), except that the destination is in DE not HL.

Entry conditions:

DE contains the address to jump to.

All registers and flags are passed to the target routine untouched.

Exit conditions:

All registers and flags are as set by the target routine.

Notes:

Calling PCDE INSTRUCTION is a useful way of invoking a routine whose address has been picked out of a table or otherwise established at run time.

Related entries:

KL FAR PCHL
KL LOW PCHL
KL SIDE PCHL
PCBC INSTRUCTION
PCHL INSTRUCTION

LOW: FAR CALL RST 3 #0018

Call subroutine in RAM or any ROM, takes inline address of 'far address'.

Action:

RST 3 is used to extend the instruction set. It is an expanded form of the CALL instruction that allows routines to be called anywhere in RAM or in any ROM. The restart is followed by the address of a 3 byte 'far address' which specifies the location to call and the required ROM state and ROM selection.

Entry conditions:

All registers and flags are passed to the target routine untouched except for IY (which is set to point at a background ROM's upper data area).

Exit conditions:

IY preserved.
All other registers and flags are as set by the target routine.

Notes:

The restart instruction takes a 2 byte inline parameter which is the address of a 'far address'. The 'far address' is laid out as follows:

Byte:	0	1	2
	Address		ROM

Bytes 0..1 give the address of the routine to call.

Byte 2 is the ROM select byte which takes values as follows:

#00..#FB:	Select the given ROM, enable upper, disable lower.
#FC:	No change of ROM selection, enable upper, enable lower.
#FD:	No change of ROM selection, enable upper, disable lower.
#FE:	No change of ROM selection, disable upper, enable lower.
#FF:	No change of ROM selection, disable upper, disable lower.

The reason that the 'far address' is not contained in the FAR CALL instruction directly is because the ROM select byte for routines in ROM will depend upon the particular configuration of expansion ROMs on the machine and must therefore be established and set at run time.

Registers are passed to the target routine untouched except for the IY register. When entering a background ROM this is set to point at the base of the ROM's upper data area. (See section 9.4 and KL INIT BACK).

The target routine returns to the instruction immediately following the inline parameter. The ROM select and ROM state are restored to what they were before the call. This is accomplished by pushing values on the stack and so care should be taken when indexing up the stack after a FAR CALL instruction. (The stack usage is 4 bytes for ROM select bytes in the range #FC..#FF and 6 bytes for ROM select bytes in the range #00..#FB.)

Interrupts are enabled.

Related entries:

KL FAR ICALL
KL FAR PCHL
LOW JUMP (RST1)
SIDE CALL (RST2)

LOW: KL FAR PCHL

Call subroutine in RAM or any ROM.
C and HL contain the 'far address' to call.

Action:

The far call mechanism allows subroutines to be called anywhere in RAM or in any ROM. This routine takes a 'far address' and calls the given routine setting the requested ROM state and ROM selection.

Entry conditions:

HL contains the address of the routine to call.
C contains the ROM select byte.

All registers and flags are passed to the target routine untouched except for IY (which is set to point at a background ROM's upper data area).

Exit conditions:

IY is preserved.
All other registers and flags are as set by the target routine.

Notes:

The ROM select byte takes values as follows:

#00..#FB:	Select the given ROM, enable upper, disable lower.
#FC:	No change of ROM selection, enable upper, enable lower.
#FD:	No change of ROM selection, enable upper, disable lower.
#FE:	No change of ROM selection, disable upper, enable lower.
#FF:	No change of ROM selection, disable upper, disable lower.

Registers are passed to the target routine untouched except for the IY index register. When entering a background ROM this is set to point at the base of the ROM's upper data area. (See section 9.4 and KL INIT BACK).

When the target routine returns, the ROM select and ROM state are restored to what they were before the call. This is accomplished by pushing values onto the stack and so care should be taken when indexing up the stack after using this routine. (The stack usage is 4 bytes for ROM select bytes in the range #FC..#FF and 6 bytes for ROM select bytes in the range #00..#FB.)

Interrupts are enabled.

Related entries:

FAR CALL (RST3)
KL FAR ICALL
KL LOW PCHL
KL SIDE PCHL

LOW: PCHL INSTRUCTION #001E

Jump to address in HL.

Action:

Entry comprises a JP (HL) instruction (or PCHL in some assembler dialects).

Entry conditions:

HL contains the address to jump to.

All registers and flags are passed to the target routine untouched.

Exit conditions:

All registers and flags are as set by the target routine.

Notes:

Calling PCHL INSTRUCTION is a useful way of invoking a routine whose address has been picked out of a table or otherwise established at run time.

Related entries:

KL FAR PCHL
KL LOW PCHL
KL SIDE PCHL
PCBC INSTRUCTION
PCDE INSTRUCTION

LOW: RAM LAM RST 4 #0020

LD A,(HL) with all ROMs disabled.

Action:

RST 4 is used to extend the instruction set. It is equivalent to a LD A, (HL) instruction except that it always reads from RAM irrespective of whether the ROMs are enabled or not.

Entry conditions:

HL contains the address of the location to read.

Exit conditions:

A contains the value read from the given location.

All other registers and flags preserved.

Notes:

Writing to a location always writes to RAM, even if the location is in one of the ROM areas and the ROM is enabled. The RAM LAM, RST 4, 'instruction' is the read equivalent.

Interrupts are enabled.

Related entries:

KL LDDR
KL LDIR

LOW: KL FAR ICALL #0023

Call subroutine in RAM or any ROM, HL points at 'far address'.

Action:

The far call mechanism allows subroutines to be called anywhere in RAM or in any ROM. This routine takes the address of a 'far address' and calls the given routine setting the ROM state and ROM selection required.

Entry conditions:

HL contains the address of the 'far address' to call.

All registers and flags are passed to the target routine untouched except for IY (which is set to point at a background ROM's upper data area).

Exit conditions:

IY is preserved.
All other registers and flags are as set by the target routine.

Notes:

The parameter passed is the address of a 3 byte 'far address'. This is laid out as follows:

Byte:	0	1	2
	Address		ROM

Bytes 0..1 give the address of the routine to call.
Byte 2 is the ROM select byte which takes values as follows:

#00..#FB:	Select the given ROM, enable upper, disable lower.
#FC:	No change of ROM selection, enable upper, enable lower.
#FD:	No change of ROM selection, enable upper, disable lower.
#FE:	No change of ROM selection, disable upper, enable lower.
#FF:	No change of ROM selection, disable upper, disable lower.

Registers are passed to the target routine untouched except for the IY index register. When entering a background ROM this is set to point at the base of the ROM's upper data area. (See section 9.4 and KL INIT BACK).

When the target routine returns, the ROM select and ROM state are restored to what they were before the call. This involves pushing values onto the stack and so care should be taken in indexing up the stack after calling this routine. (The stack usage is 4 bytes for ROM select bytes in the range #FC..#FF and 6 bytes for ROM select bytes in the range #00..#FB.)

Interrupts are enabled.

Related entries:

KL FAR CALL
KL FAR PCHL

LOW: FIRM JUMP

RST 5 #0028

Jump to lower ROM, takes inline address to jump to.

Action:

RST 5 is used to extend the instruction set. It is an expanded form of the jump instruction for jumping to routines in the lower ROM or into the central 32K of RAM. The restart is followed by the address of the routine to jump to.

Entry conditions:

All registers and flags are passed to the target routine untouched.

Exit conditions:

All registers and flags are as set by the target routine.

Notes:

The lower ROM is enabled before the jump is taken and is disabled (rather than restored) when the target routine returns. Neither the upper ROM state nor the ROM selection are changed. Two bytes are pushed onto the stack and so care should be taken when indexing up the stack (to find the address of inline parameters, for example).

It is assumed that the destination of the jump is a routine which will return in the usual way. The restart instruction itself does not return. The value at top of stack when a FIRM JUMP is executed must, therefore, be a return address.

The FIRM JUMP, RST 5, 'instruction' may replace the first byte of a JP (jump) instruction, particularly in jumpblocks, much like a LOW JUMP. A FIRM JUMP is slightly faster than a LOW JUMP but a LOW JUMP is more flexible in dealing with ROM states.

Interrupts are enabled.

Related entries:

LOW JUMP (RST1)

LOW: USER RESTART RST 6 #0030

Undedicated RST instruction.

Action:

The eight bytes from #0030 to #0037 inclusive may be patched as required.

Entry conditions:

Unknown.

Exit conditions:

Unknown.

Notes:

If the lower ROM is disabled when an RST 6 instruction is executed then the instructions patched into locations #0030 to #0037 are executed in the normal way.

If the lower ROM is enabled when the RST 6 instruction is executed then the firmware disables the lower ROM and jumps to #0030 to execute the instructions planted by the user.

Generally the lower ROM is disabled except while the firmware is active. Since there are no RST 6s in the firmware there should be no problem about the ROM state when a RST 6 is executed. However, to cope with all eventualities, if the lower ROM is found to be enabled when the restart is executed then the ROM state before the lower ROM is disabled is saved in location #002B. If the lower ROM is found to be disabled then location #002B is left untouched. The value stored is suitable to be passed to KL ROM RESTORE to re-enable the ROM (although KL L ROM ENABLE will have the same effect).

The user can detect whether the lower ROM was enabled when the restart was executed if location #002B is set to zero when the RST 6 area is patched and after processing each restart. If #002B is zero when the RST 6 code is entered then the lower ROM was disabled, and if it is non-zero then the lower ROM was enabled.

The default action for RST 6 as set at power-up is to perform a RST 0, i.e. a system reset.

Related entries:

None.

LOW: INTERRUPT ENTRY RST 7 #0038

Hardware interrupt entry point.

Action:

The Z80 runs in interrupt mode 1, which treats normal interrupts as RST 7 instructions. The firmware interrupt handler looks after the built in regular time interrupt. External interrupts, generated by expansion hardware, are passed on to user supplied software.

Entry conditions:

No conditions.

Exit conditions:

All registers and flags preserved.

Notes:

The user must not use RST 7s as these are dedicated to the processing of interrupts.

If the interrupt is from an external source then the user supplied interrupt routine, EXT INTERRUPT, is called.

See section 10 for a fuller discussion of interrupts.

The user may patch this area (#0038..#003A inclusive) to intercept interrupts if it is absolutely necessary (see Appendix XI).

Related entries:

EXT INTERRUPT

LOW: EXT INTERRUPT #003B

External interrupt routine.

Action:

The five bytes from #003B to #003F inclusive must be patched by the user if there are going to be any external interrupts. When an external interrupt is detected by the firmware interrupt handler the lower ROM is disabled and the code at #003B is called.

Entry conditions:

No conditions.

Exit conditions:

AF, BC, DE and HL corrupt.
All other registers preserved.

Notes:

When the routine is called interrupts are disabled and they must remain disabled. Under no circumstances may the user enable interrupts or use the second register set. Before the routine returns it must clear the interrupt source.

See section 10.2 for a discussion of external interrupts.

When an interrupt routine is set up the current contents of #003B..#003F should be copied elsewhere before they are replaced. If, when the routine is called, it discovers that its hardware is not responsible for the interrupt then it should jump to the copy of the previous external interrupt routine (whose hardware may be responsible).

The purpose of an interrupt routine is to clear the interrupt as quickly as possible, and perhaps perform a minimum of processing. While in the interrupt path no further interrupts are acknowledged. If the interrupt generates a substantial work load, then it should be translated into an event, so that the system is not delayed in the interrupt path for any longer than necessary (see section 10.3).

The interrupt routine must be in RAM at addresses lower than #C000 (as the ROM enable and disable routines cannot be called from the interrupt path).

The default external interrupt routine merely returns. This means that the interrupt will not be cleared and so it will repeat as soon as interrupts are re-enabled. This will cause the machine to 'lock up'.

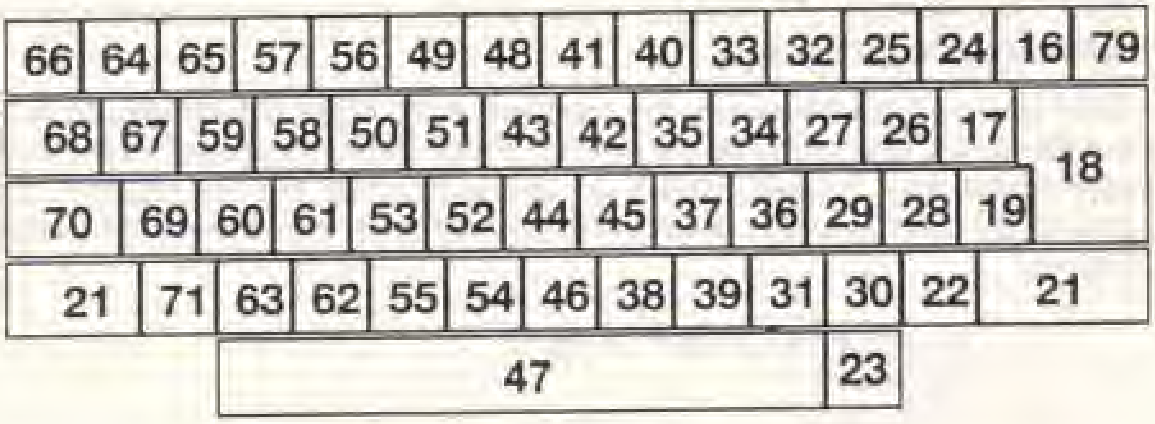
Related entries:

INTERRUPT ENTRY
KL EVENT

Appendix I
Key Numbering.

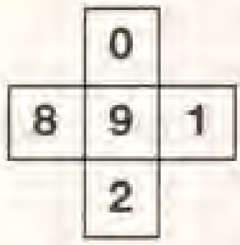
The various tables in the Key Manager, such as the translation tables or the repeating key table, are all accessed by key number. The numbering of the keys (and joysticks) is given in the following diagrams:

Main Keyboard



Function/Numeric Keypad

Cursor Keys



Note that joystick 1 overlays keys 48..53 on the main keyboard and is indistinguishable from them.

The following table translates key numbers in the opposite direction, from the key number to the inscription on the keytop. Where there is a symbol on the keytop an appropriate word is used, RIGHT for the right cursor key for example. Brackets around the inscription are used to distinguish the various areas of the keyboard as follows:

- {..} Function key (numeric keypad).
- (..) Joystick 0.
- [..] Joystick 1.

	0	1	2	3	4	5	6	7
0	UP	RIGHT	DOWN	{9}	{6}	{3}	{ENTER}	{.}
8	LEFT	COPY	{7}	{8}	{5}	{1}	{2}	{0}
16	CLR	[	ENTER	]	{4}	SHIFT	\	CTRL
24	↑	-	@	P	;	:	/	.
32	0	9	O	I	L	K	M	,
40	8	7	U	Y	H	J	N	SPACE
48	6	5	R	T	G	F	B	V
	[UP]	[DOWN]	[LEFT]	[RIGHT]	[FIRE2]	[FIRE1]	[SPARE]	
56	4	3	E	W	S	D	C	X
64	1	2	ESC	Q	TAB	A	CAPS	Z
72	(UP)	(DOWN)	(LEFT)	(RIGHT)	(FIRE2)	(FIRE1)	(SPARE)	DEL

Appendix II
Key Translation Tables.

See section 3, and section 3.2 in particular, for a description of key translation. Also, Appendix I, which gives the key numbering scheme, may be of interest.

There are three keyboard translation tables used. These convert a key into its associated character or token. One table is used to translate keys when the control key is pressed, one is used to translate keys when the shift key is pressed or the shift lock is on but the control key is not pressed, the last is used to translate keys when neither shift nor control is pressed.

The diagrams following describe the default translation tables. Where possible the correct character has been placed on the key. The actual value for each of these characters can be found in Appendix VI on the character set. In the cases where the key produces a value which is not a printable ASCII character the abbreviations in the following table will be used. The default settings of the expansion tokens are given in Appendix IV.

Characters and Codes.

NUL	#00	ASCII control code.
SOH	#01	ASCII control code.
STX	#02	ASCII control code.
ETX	#03	ASCII control code.
EOT	#04	ASCII control code.
ENQ	#05	ASCII control code.
ACK	#06	ASCII control code.
BEL	#07	ASCII control code.
BS	#08	ASCII control code.
HT	#09	ASCII control code.
LF	#0A	ASCII control code.
VT	#0B	ASCII control code.
FF	#0C	ASCII control code.
CR	#0D	ASCII control code.
SO	#0E	ASCII control code.
SI	#0F	ASCII control code.

DLE	#10	ASCII control code.
DC1	#11	ASCII control code.
DC2	#12	ASCII control code.
DC3	#13	ASCII control code.
DC4	#14	ASCII control code.
NAK	#15	ASCII control code.
SYN	#16	ASCII control code.
ETB	#17	ASCII control code.
CAN	#18	ASCII control code.
EM	#19	ASCII control code.
SUB	#1A	ASCII control code.
ESC	#1B	ASCII control code.
FS	#1C	ASCII control code.
GS	#1D	ASCII control code.
RS	#1E	ASCII control code.
US	#1F	ASCII control code.
SPACE	#20	ASCII space character.
UP	#5E	Up arrow.
DEL	#7F	ASCII code.
LB	#A3	Pound character.

Expansion Tokens.

F0	#80	Function key 0.
F1	#81	Function key 1.
F2	#82	Function key 2.
F3	#83	Function key 3.
F4	#84	Function key 4.
F5	#85	Function key 5.
F6	#86	Function key 6.
F7	#87	Function key 7.
F8	#98	Function key 8.
F9	#89	Function key 9.

F.	#8A	Function key full stop.
FEN	#8B	Function key enter without control pressed.
FRUN	#8C	Function key enter with control pressed.

Edit and Cursor Codes.

COPY	#E0	Copy key.
INS	#E1	Insert/overwrite toggle key.
WUP	#F0	Write cursor up.
WDN	#F1	Write cursor down.
WLT	#F2	Write cursor left.
WRT	#F3	Write cursor right.
RUP	#F4	Read cursor up.
RDN	#F5	Read cursor down.
RLT	#F6	Read cursor left.
RRT	#F7	Read cursor right.
BEG	#F8	Write cursor to start of text.
END	#F9	Write cursor to end of text.
STA	#FA	Write cursor to start of line.
FIN	#FB	Write cursor to end of line.

System Tokens.

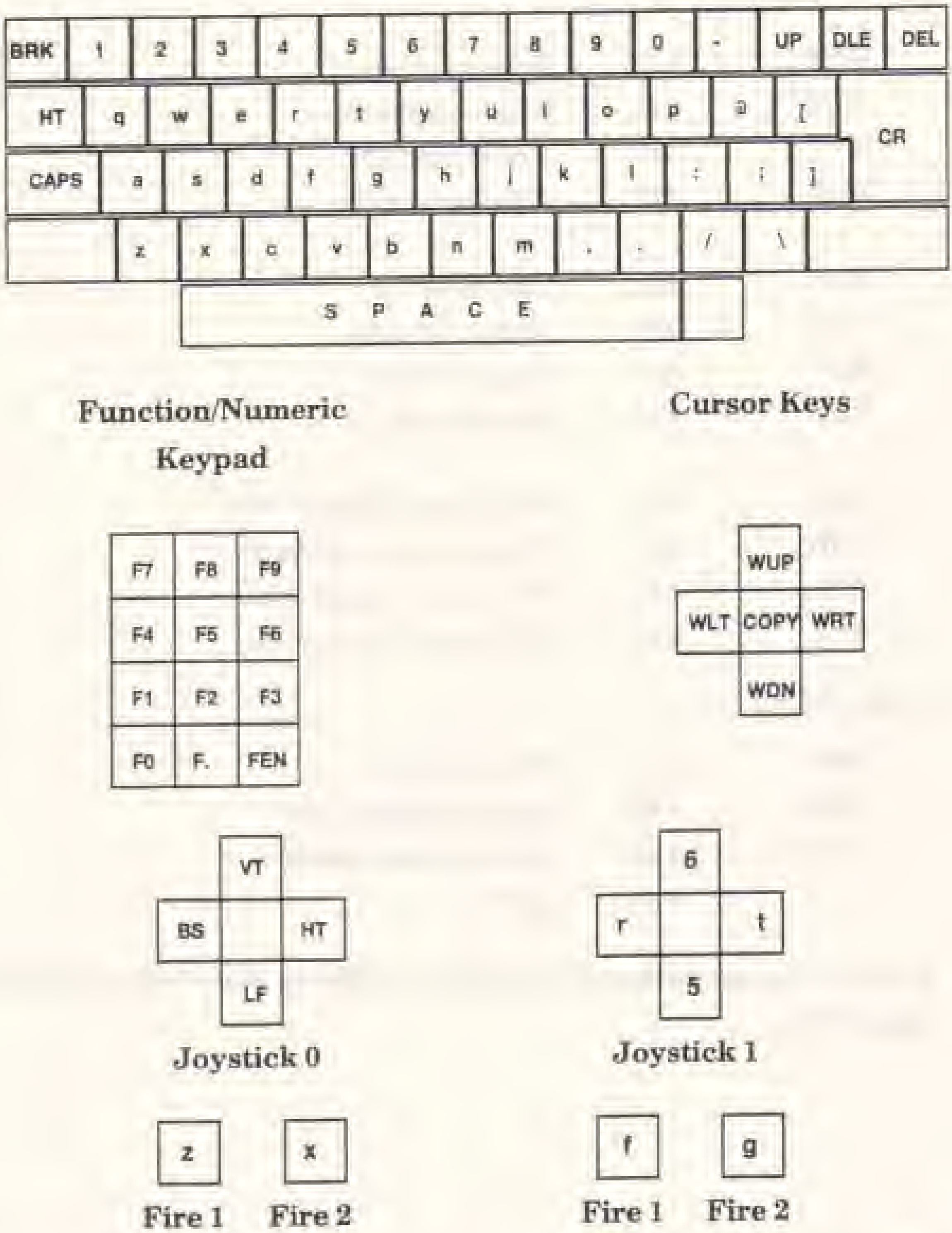
BRK	#FC	Break key token.
CAPS	#FD	Caps lock toggle token.
SHIFT	#FE	Shift lock toggle token.
	#FF	Ignore.

Keys that are not marked in the diagrams following generate the system ignore token, #FF.

Normal Translation Table.

The following diagram describes the translation when neither shift nor control is pressed.

Main Keyboard



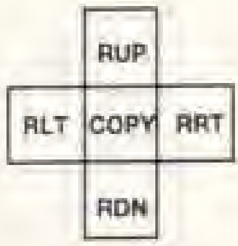
Shift Translation Table.

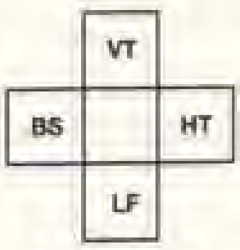
The following diagram describes the translation when either shift key is pressed, or the shift lock is on, but the control key is not pressed.

Main Keyboard

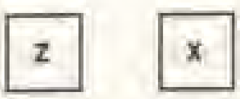
Function/Numeric Keypad

Cursor Keys

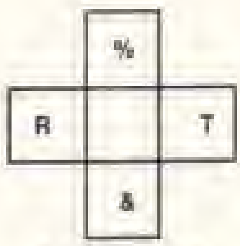




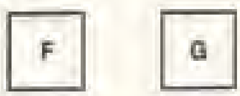
Joystick 0



Fire 1 **Fire 2**



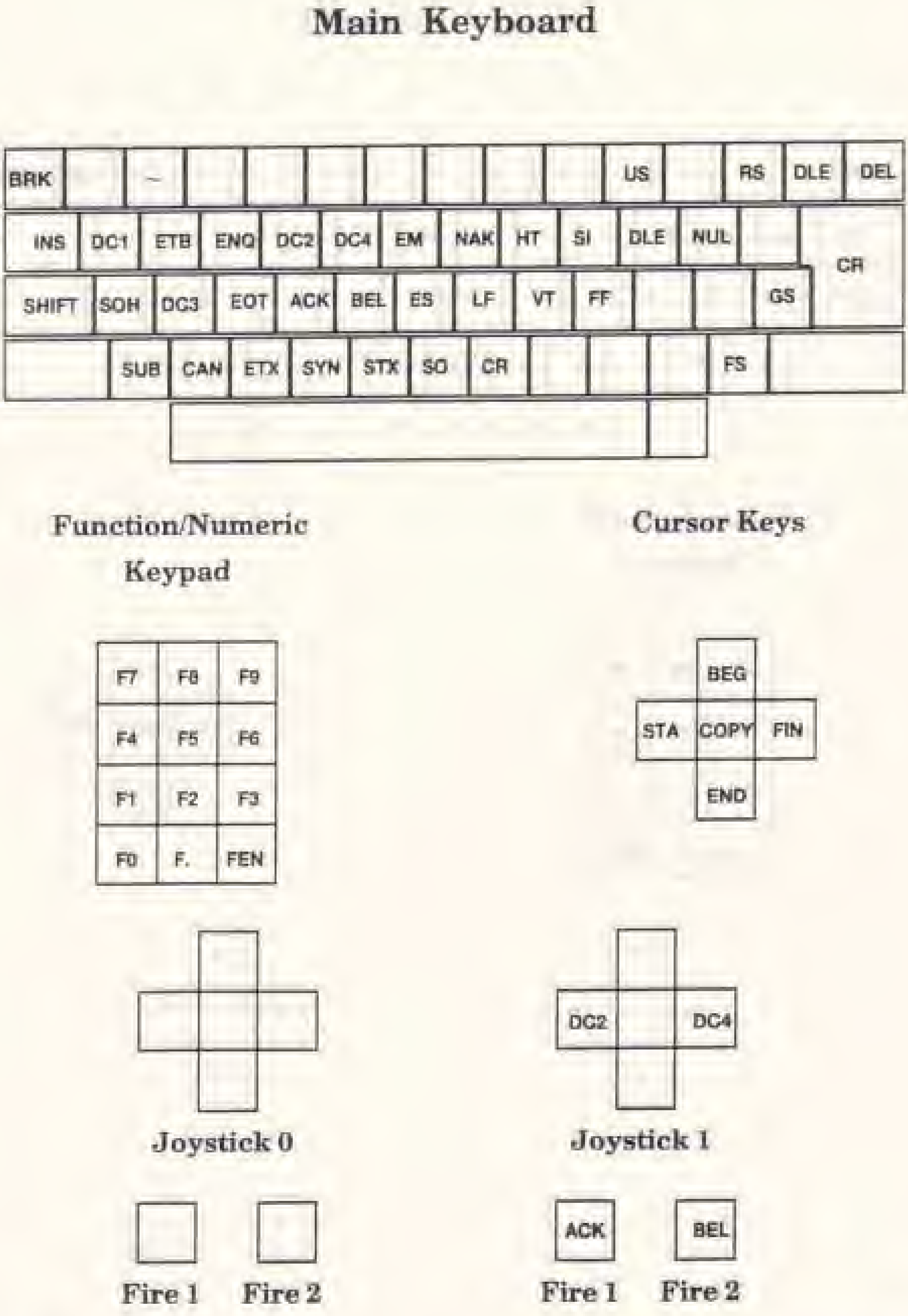
Joystick 1



Fire 1 **Fire 2**

Control Translation Table.

The following diagram describes the translation when the control key is pressed.

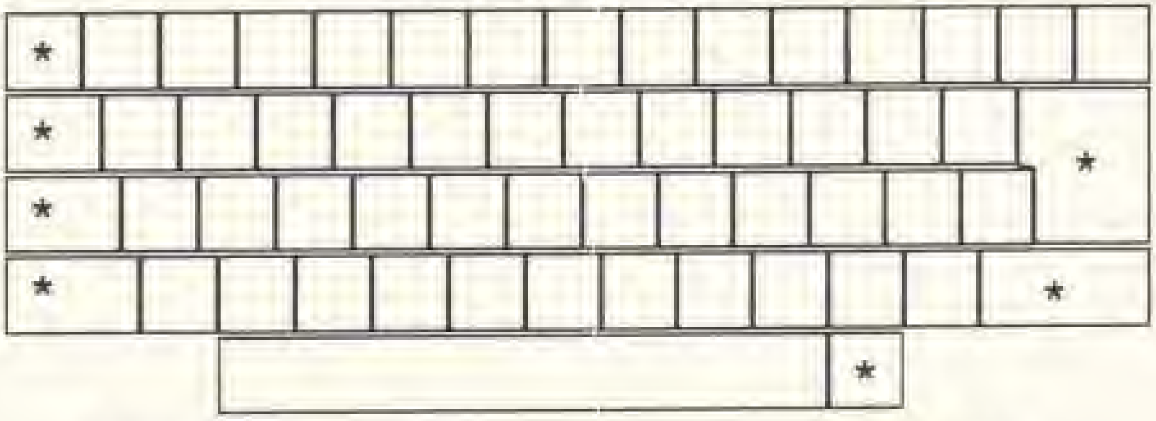


Appendix III
Repeating Keys.

Which keys are allowed to repeat may be set by the user. See section 3 (and section 3.5 in particular) for a full description of repeating keys. Also, see Appendix I which gives the key numbering scheme.

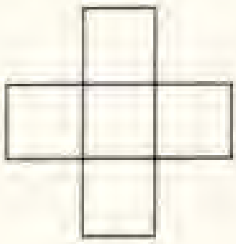
The default repeating key table is described in the following diagrams. Keys which are not allowed to repeat are marked with an asterisk.

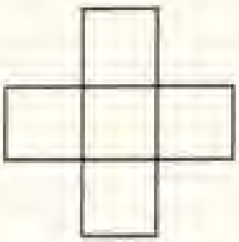
Main Keyboard



Function/Numeric Keypad

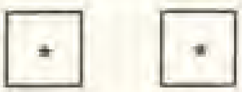
Cursor Keys

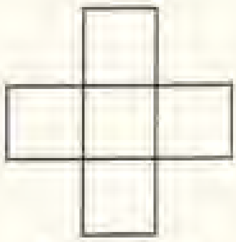




Joystick 0

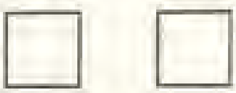
Fire 1





Joystick 1

Fire 1



Fire 2

Appendix IV
Function Keys and Expansion Strings.

Function keys are more fully explained in section 3, and in section 3.7 in particular. The following table specifies the default string for each expansion token and which key the token is associated with by default.

Token	Value	Default String	Default Key
0	#80	0	Function key 0.
1	#81	1	Function key 1.
2	#82	2	Function key 2.
3	#83	3	Function key 3.
4	#84	4	Function key 4.
5	#95	5	Function key 5.
6	#86	6	Function key 6.
7	#87	7	Function key 7.
8	#88	8	Function key 8.
9	#89	9	Function key 9.
10	#8A	.	Function key full stop.
11	#8B	CR	Function key enter.
12	#8C	R U N " CR	Function key enter with control
13..31	#8D..#9F		None.

Tokens 13..31 are all set to empty strings and none of them are defaulted to associate with a key.

CR stands for carriage return (character #OD)

Appendix V
Inks and Colours.

A full discussion of inks and colours can be found in section 6.2. This appendix lists the colours that are available and the default settings for the inks

There are 27 colours available. The Screen Pack refers to these colours by a grey scale number so that colour 0 is the darkest and colour 26 is the brightest. The hardware requires these grey scales to be translated into the hardware code for the colour. It is unlikely that the user will ever need to deal with the hardware numbers, they are merely given for information.

Grey Scale	Colour	HW Number
0	Black	20
1	Blue	4
2	Bright blue	21
3	Red	28
4	Magenta	24
5	Mauve	29
6	Bright red	12
7	Purple	5
8	Bright magenta	13
9	Green	22
10	Cyan	6
11	Sky blue	23
12	Yellow	30
13	White	0
14	Pastel blue	31
15	Orange	14
16	Pink	7
17	Pastel magenta	15
18	Bright green	18
19	Sea green	2
20	Bright cyan	19
21	Lime	26
22	Pastel green	25
23	Pastel cyan	27
24	Bright yellow	10
25	Pastel yellow	3
26	Bright white	11

The user can set the colours in which the 16 inks and the border are displayed. The following table gives the default settings:

Ink	Colour	Colour Numbers
Border	Blue	(1/1)
0	Blue	(1/1)
1	Bright yellow	(24/24)
2	Bright cyan	(20/20)
3	Bright red	(6/6)
4	Bright white	(26/26)
5	Black	(0/0)
6	Bright blue	(2/2)
7	Bright magenta	(8/8)
8	Cyan	(10/10)
9	Yellow	(12/12)
10	Pastel blue	(14/14)
11	Pink	(16/16)
12	Bright green	(18/18)
13	Pastel green	(22/22)
14	Flashing blue / bright yellow	(1/24)
15	Flashing sky blue / pink	(11/16)

Appendix VI
Displayed Character Set.

There are 256 symbols in the displayed character set. All of these can be printed, although it requires special effort to print characters 0..31 which are often interpreted as control codes. The user can set the matrix for any or all characters (see section 4.6). The following lists describe the default character set.

The character set is split into a number of areas for ease of description:

0..31	(#00..#1F)	ASCII control codes.
32..127	(#20..#7F)	ASCII characters.
128..143	(#80..#8F)	Block graphics.
144..159	(#90..#9F)	Line graphics.
160..191	(#A0..#BF)	Further characters.
192..255	(#C0..#FF)	Miscellaneous graphic symbols.

a. ASCII Control Codes.

0	#00	NUL	□	Square.
1	#01	SOH	┌	Upside down L.
2	#02	STX	⊥	Upside down T.
3	#03	ETX	┘	Backwards L.
4	#04	EOT	ϟ	Lightning flash.
5	#05	ENQ	⊠	Square with a diagonal cross.
6	#06	ACK	✓	Tick.
7	#07	BEL	🔔	Bell (semi-circle with feet).
8	#08	BS	←	Left pointing arrow.
9	#09	HT	→	Right pointing arrow.
10	#0A	LF	↓	Down pointing arrow.
11	#0B	VT	↑	Up pointing arrow.
12	#0C	FF	⇓	Christmas tree (down pointing arrow with a tail).
13	#0D	CR	↵	Bent left pointing arrow.
14	#0E	SO	⊗	Circle with a diagonal cross.
15	#0F	SI	⊙	Circle with a central dot.

16	#10	DLE		Square with a horizontal bar.
17	#11	DC1		Circle with three o'clock.
18	#12	DC2		Circle with half past three.
19	#13	DC3		Circle with half past nine.
20	#14	DC4		Circle with nine o'clock.
21	#15	NAK		Crossed out tick.
22	#16	SYN		Square wave.
23	#17	ETB		Sideways T.
24	#18	CAN		Hour glass.
25	#19	EM		Vertical bar with a central blob.
26	#1A	SUB		Backwards question mark.
27	#1B	ESC		Circle with a horizontal bar.
28	#1C	FS		Square with nine o'clock.
29	#1D	GS		Square with half past nine.
30	#1E	RS		Square with half past three.
31	#1F	US		Square with three o'clock.

b. ASCII Characters.

Characters 32..127 (#20..#7F) are listed in the following table. They make up the standard ASCII character set.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
32 #20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48 #30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64 #40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80 #50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	_
96 #60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112 #70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

c. Block Graphics.

Characters 128..143 (#80..#8F) are a set of block graphics. Each character is divided into four cells. Bits 0..3 of the character number determine which cells are filled. If the appropriate bit is set then the cell is filled in, otherwise it is left blank. The cells are:

Bit 0	Bit 1
Bit 2	Bit 3

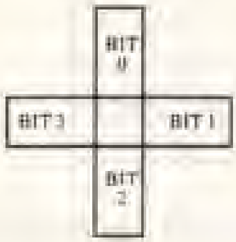
In the following list cells that are filled are marked with an ■, cells that are blank are marked with an □.

128	#80	Block graphic	⊞
129	#81	Block graphic	▘
130	#82	Block graphic	▝
131	#83	Block graphic	▀
132	#84	Block graphic	▖
133	#85	Block graphic	▌
134	#86	Block graphic	▞
135	#87	Block graphic	▛
136	#88	Block graphic	▗
137	#89	Block graphic	▚
138	#8A	Block graphic	▐
139	#8B	Block graphic	▜
140	#8C	Block graphic	▄
141	#8D	Block graphic	▙
142	#8E	Block graphic	▟
143	#8F	Block graphic	█

d. Line Graphics

Characters 144..159 (#90..#9F) are a set of line graphics. The lines join the centre of the character to the centre of an edge. Each of the lines is associated with a bit of the character number. If the bit is set then the line is present, if the bit is not set then the line is not present. The central block of the character is always set.

The lines are associated with bits as follows:



144	#90	Line graphic	·
145	#91	Line graphic	╵
146	#92	Line graphic	╶
147	#93	Line graphic	└
148	#94	Line graphic	╷
149	#95	Line graphic	│
150	#96	Line graphic	┌
151	#97	Line graphic	├
152	#98	Line graphic	╴
153	#99	Line graphic	┘
154	#9A	Line graphic	─
155	#9B	Line graphic	┴
156	#9C	Line graphic	┐
157	#9D	Line graphic	┤
158	#9E	Line graphic	┬
159	#9F	Line graphic	┼

e. Further Characters.

160	#A0	^	Circumflex
161	#A1	´	Acute accent
162	#A2	¨	Umlaut
163	#A3	£	Pound
164	#A4	©	Copyright
165	#A5	¶	Pilcrow
166	#A6	§	Section
167	#A7	‘	Open single quote (pairs with character 39)
168	#A8	¼	One quarter
169	#A9	½	One half
170	#AA	¾	Three quarters
171	#AB	±	Plus or minus
172	#AC	÷	Division
173	#AD	¬	Not
174	#AE	¿	Inverted question mark
175	#AF	¡	Inverted exclamation mark
176	#B0	α	Lower case alpha
177	#B1	β	Lower case beta
178	#B2	γ	Lower case gamma
179	#B3	δ	Lower case delta

180	#B4	ε	Lower case epsilon
181	#B5	θ	Lower case theta
182	#B6	λ	Lower case lambda
183	#B7	μ	Lower case mu
184	#B8	π	Lower case pi
185	#B9	σ	Lower case sigma
186	#BA	φ	Lower case phi
187	#BB	ψ	Lower case psi
188	#BC	χ	Lower case chi
189	#BD	ω	Lower case omega
190	#BE	Σ	Upper case sigma
191	#BF	Ω	Upper case omega

f. Miscellaneous Graphics Symbols.

192	#C0	╱	Diagonal line joining top to left.
193	#C1	╲	Diagonal line joining top to right.
194	#C2	╱	Diagonal line joining bottom to right.
195	#C3	╲	Diagonal line joining bottom to left.
196	#C4	∧	Diagonal lines joining top to left and right.
197	#C5	〉	Diagonal lines joining right to top and bottom.
198	#C6	∨	Diagonal lines joining bottom to right and left.
199	#C7	〈	Diagonal lines joining left to top and bottom.
200	#C8	╱╱	Diagonal lines joining top to left and bottom to right.
201	#C9	╲╲	Diagonal lines joining top to right and bottom to left.
202	#CA	◇	Diamond joining all edges.

203	#CB	Both major diagonals (large X).
204	#CC	Forwards major diagonal (large slash).
205	#CD	Backwards major diagonal (large backslash).
206	#CE	Chequered pattern.
207	#CF	Shading.
208	#D0	Line along top edge.
209	#D1	Line along right edge.
210	#D2	Line along bottom edge.
211	#D3	Line along left edge.
212	#D4	Triangle filling top left corner.
213	#D5	Triangle filling top right corner.
214	#D6	Triangle filling bottom right corner.
215	#D7	Triangle filling bottom left corner.
216	#D8	Top half shaded.
217	#D9	Right half shaded.
218	#DA	Bottom half shaded.
219	#DB	Left half shaded.
220	#DC	Shaded triangle filling top left corner.
221	#DD	Shaded trinagle filling top right corner.
222	#DE	Shaded triangle filling bottom right corner.
223	#DF	Shaded triangle filling bottom left corner.
224	#E0	Happy face.
225	#E1	Sad face.
226	#E2	Club.

227	#E3	Diamond.
228	#E4	Heart.
229	#E5	Spade.
230	#E6	Empty circle.
231	#E7	Filled circle.
232	#E8	Empty square.
233	#E9	Filled square.
234	#EA	Male (Mars).
235	#EB	Female (Venus).
236	#EC	Crochet.
237	#ED	Quaver.
238	#EE	Star.
239	#EF	Rocket.
240	#F0	Up pointing arrow head.
241	#F1	Down pointing arrow head.
242	#F2	Left pointing arrow head.
243	#F3	Right pointing arrow head.
244	#F4	Up pointing triangle.
245	#F5	Down pointing triangle.
246	#F6	Right pointing triangle.
247	#F7	Left pointing triangle.
248	#F8	Dancing person standing.
249	#F9	Dancing person doing splits.
250	#FA	Dancing person with left leg out.

251	#FB		Dancing person with right leg out.
252	#FC		Bomb.
253	#FD		Mushroom (cloud).
254	#FE		Up and down arrow.
255	#FF		Right and left arrow.

Appendix VII
Text VDU Control Codes.

Character values in the range 0..31 sent to the main Text VDU output routine (TXT OUTPUT) do not produce a character on the screen, but are interpreted as control codes. These codes may affect the meaning of one or more of the following characters, which are the code's parameters.

All control codes work on the currently selected stream unless otherwise indicated. For instance, setting the pen, code 15, sets the text pen ink for the currently selected stream whilst setting the colour of an ink, code 28, will affect all streams (and the Graphics VDU).

Certain codes force the current position (the cursor position) to a legal position inside the current window before they are obeyed. This is explained in more detail in section 4.5. The cursor may be left in an illegal position.

The following table specifies the default actions for the control codes. By changing entries in the control code table the action of these codes can be altered as desired. See section 4.7 for a full description.

Code	Name	Params	Action
0	NUL	0	No effect.
1	SOH	1	Print the character given by the parameter (see TXT WR CHAR). This allows characters 0..31 to be printed.
2	STX	0	Disable the cursor blob (see TXT CUR DISABLE). Reverses the effect of ETX (code 3).
3	ETX	0	Enable the cursor blob (see TXT CUR ENABLE). Reverses the effect of STX (code 2).
4	EOT	1	Set the screen mode given by the parameter (see SCR SET MODE). The parameter is taken MOD 4 and the value 3 is ignored: 0 sets mode 0 (160 x 200). 1 sets mode 1 (320 x 200). 2 sets mode 2 (640 x 200).
5	ENQ	1	Print the character given by the parameter using the Graphics VDU as if the graphic character write mode was active (see TXT SET GRAPHIC and GRA WR CHAR).

6	ACK	0	Enable the VDU (see TXT VDU ENABLE). Reverses the effect of NAK (code 21).
7	BEL	0	Makes a short bleep sound. Note that this flushes the sound queues.
8	BS	0	Make the current position legal then move left one character.
9	TAB	0	Make the current position legal then move right one character.
10	LF	0	Make the current position legal then move down one line.
11	VT	0	Make the current position legal then move up one line.
12	FF	0	Clear the current window and move the current position to the top left corner (see TXT CLEAR WINDOW).
13	CR	0	Make the current position legal and then move it to the left edge of the window on the current line (see TXT SET COLUMN).
14	SO	1	Set the paper ink to the ink given by the parameter (see TXT SET PAPER). Parameter is taken MOD 16.
15	SI	1	Set the pen ink to the ink given by the parameter (see TXT SET PEN). Parameter is taken MOD 16.
16	DLE	0	Make the current position legal then clear it to the current paper ink.
17	DC1	0	Make the current position legal then clear from the left edge of the window to the current position inclusive. The affected cells are set to the current paper ink.
18	DC2	0	Make the current position legal then clear from it to the right edge of the window inclusive. The affected cells are set to the current paper ink.
19	DC3	0	Make the current position legal then clear from the start of the window to the current position inclusive. The affected cells are set to the current paper ink.
20	DC4	0	Make the current position legal then clear from it to the end of the window inclusive. The affected cells are set to the current paper ink.
21	NAK	0	Disable the VDU (see TXT VDU DISABLE). Reverses the effect of ACK (code 6).

22	SYN	1	Set the character write mode from the parameter (see TXT SET BACK). The parameter is taken MOD 2 and: 0 sets opaque mode (the default mode). 1 sets transparent mode.
23	ETB	1	Set the Graphics VDU write mode from the parameter (see SCR ACCESS). The parameter is taken MOD 4 and: 0 sets FORCE mode (the default mode). 1 sets XOR mode. 2 sets AND mode. 3 sets OR mode.
24	CAN	0	Exchange the current pen and paper inks (see TXT INVERSE).
25	EM	9	Set the matrix for a character (see TXT SET MATRIX). The first parameter specifies which character is to be set. The next 8 parameters are the matrix for the character (given top to bottom). If the character is not user definable then no action is taken.
26	SUB	4	Set the limits of the text window (see TXT WIN ENABLE). The first two parameters specify the left and right columns of the window (the smaller is the left column); the last two parameters specify the top and bottom rows of the window (the smaller is the top row).
27	ESC	0	No effect - available for user.
28	FS	3	Set the colours in which to display an ink (see SCR SET INK). The first parameter is taken MOD 16 and specifies which ink is to be set. The second and third parameters are taken MOD 32 and specify the two colours for the ink.
29	GS	2	Set the colours with which to display the border (see SCR SET BORDER). The two parameters are taken MOD 32 and specify the two colours for the border.
30	RS	0	Move the current position to the top left corner of the window (see TXT SET CURSOR).
31	US	2	Move the current position to a given position in the current window (see TXT SET CURSOR). The first parameter specifies the column to move to, the second parameter specifies the row to move to (row 1, column 1 is the top left corner of the window).

Appendix VIII
Notes and Tone Periods.

The tables which follow give the recommended tone period settings for notes in the even tempered scale for the full eight octave range. The period is calculated from the note frequency as follows (since the period is given in 8 microsecond units):

Period = 125000 / Frequency

The frequency for each note is calculated from International A as follows:

Frequency = 440 * (2 (Octave + (N - 10) / 12))

where:

Octave — is the octave number. 0 is the octave including International A (and middle C), -1 is the octave below, +1 is the octave above etc.

N — is the note number. 1 is C, 2 is C#, 3 is D etc.

The period is an integer value and so the frequency of the note produced is not exactly the required frequency. The relative error is given in the tables below. This is calculated as follows:

Error = (Required frequency - Actual frequency) / Required frequency

Note	Frequency	Period		Error	Octave -3
C	32.703	3822	#0EEE	-0.007%	
C#	34.648	3608	#0E18	+0.007%	
D	36.708	3405	#0D4D	-0.007%	
D#	38.891	3214	#0C8E	-0.004%	
E	41.203	3034	#0BDA	+0.009%	
F	43.654	2863	#0B2F	-0.016%	
F#	46.249	2703	#0A8F	+0.009%	
G	48.999	2551	#09F7	-0.002%	
G#	51.913	2408	#0968	+0.005%	
A	55.000	2273	#08E1	+0.012%	
A#	58.270	2145	#0861	-0.008%	
B	61.735	2025	#07E9	+0.011%	

Note	Frequency	Period		Error	Octave -2
C	65.406	1911	#0777	-0.007%	
C#	69.296	1804	#070C	+0.007%	
D	73.416	1703	#06A7	+0.022%	
D#	77.782	1607	#0647	-0.004%	
E	82.407	1517	#05ED	+0.009%	
F	87.307	1432	#0598	+0.019%	
F#	92.499	1351	#0547	-0.028%	
G	97.999	1276	#04FC	+0.037%	
G#	103.826	1204	#04D4	+0.005%	
A	110.000	1136	#0470	-0.032%	
A#	116.541	1073	#0431	+0.039%	
B	123.471	1012	#03F4	-0.038%	

Note	Frequency	Period		Error	Octave -1
C	130.813	956	#03DC	+0.046%	
C#	138.591	902	#0386	+0.007%	
D	146.832	851	#0353	-0.037%	
D#	155.564	804	#0324	+0.058%	
E	164.814	758	#02F6	-0.057%	
F	174.614	716	#02CC	+0.019%	
F#	184.997	676	#02A4	+0.046%	
G	195.998	638	#027E	+0.037%	
G#	207.652	602	#025A	+0.005%	
A	220.000	568	#0238	-0.032%	
A#	233.082	536	#0218	-0.055%	
B	246.942	506	#01FA	-0.038%	

Note	Frequency	Period		Error	Octave 0
C	261.626	478	#01DE	+0.046%	Middle C
C#	277.183	451	#01C3	+0.007%	
D	293.665	426	#01AA	+0.081%	
D#	311.127	402	#0192	+0.058%	
E	329.628	379	#017B	-0.057%	
F	349.228	358	#0166	+0.019%	
F#	369.994	338	#0152	+0.046%	
G	391.995	319	#013F	+0.037%	
G#	415.305	301	#012D	+0.005%	
A	440.000	284	#011C	-0.032%	International A
A#	466.164	268	#010C	-0.055%	
B	493.883	253	#00FD	-0.038%	

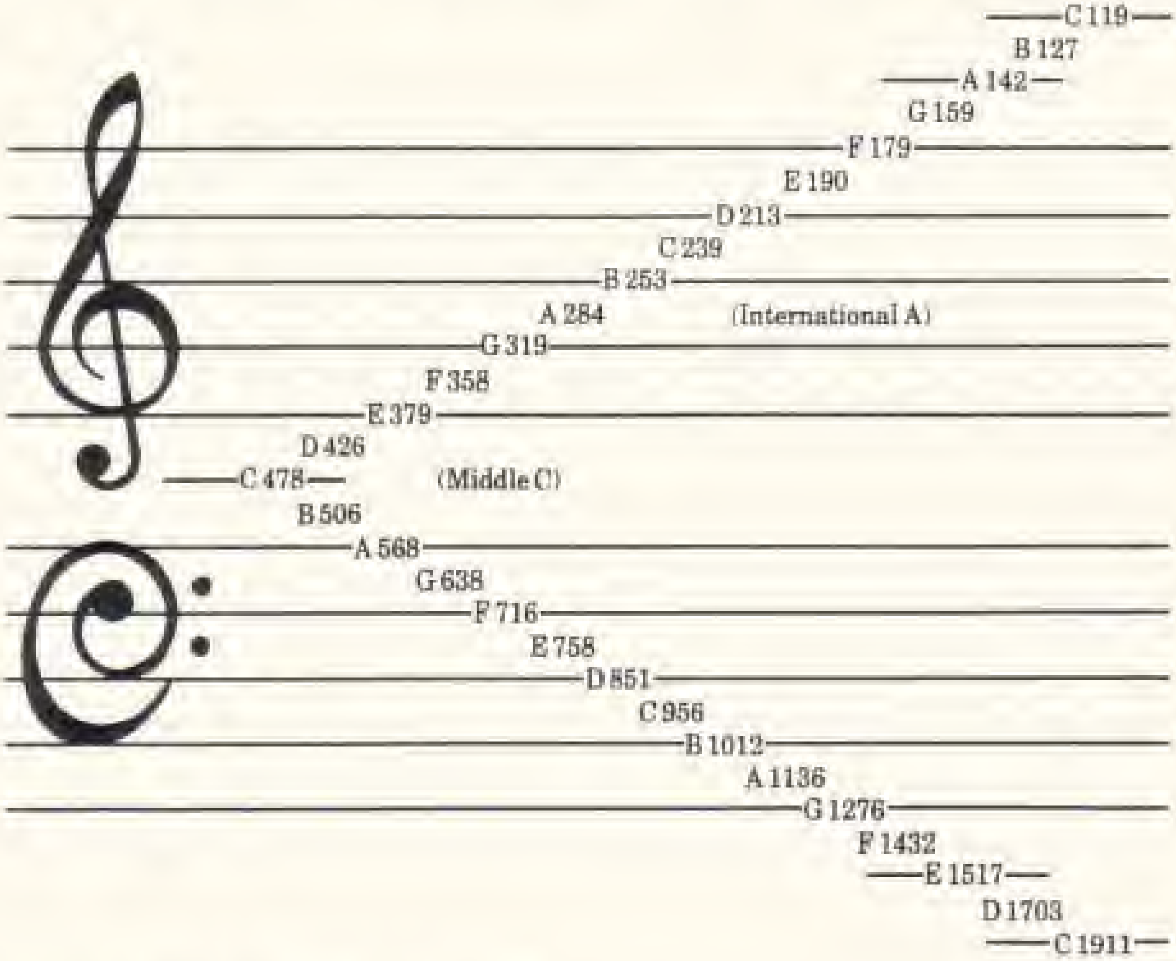
Note	Frequency	Period		Error	Octave 1
C	523.251	239	#00EF	+0.046%	
C#	554.365	225	#00E1	-0.215%	
D	587.330	213	#00D5	+0.081%	
D#	622.254	201	#00C9	+0.058%	
E	659.255	190	#00BE	+0.206%	
F	698.457	179	#00B3	+0.019%	
F#	739.989	169	#00A9	+0.046%	
G	783.991	159	#009F	-0.277%	
G#	830.609	150	#0096	-0.328%	
A	880.000	142	#008E	-0.032%	
A#	932.328	134	#0086	-0.055%	
B	987.767	127	#007F	+0.356%	

Note	Frequency	Period		Error	Octave 2
C	1046.502	119	#0077	-0.374%	
C#	1108.731	113	#0071	+0.229%	
D	1174.659	106	#006A	-0.390%	
D#	1244.508	100	#0064	-0.441%	
E	1318.510	95	#005F	+0.206%	
F	1396.913	89	#0059	-0.543%	
F#	1479.978	84	#0054	-0.548%	
G	1567.982	80	#0050	+0.350%	
G#	1661.219	75	#004B	-0.328%	
A	1760.000	71	#0047	-0.032%	
A#	1864.655	67	#0043	-0.055%	
B	1975.533	63	#003F	-0.435%	

Note	Frequency	Period		Error	Octave 3
C	2093.004	60	#003C	+0.462%	
C#	2217.461	56	#0038	-0.662%	
D	2349.318	53	#0035	-0.390%	
D#	2489.016	50	#0032	-0.441%	
E	2637.021	47	#002F	-0.855%	
F	2793.826	45	#002D	+0.574%	
F#	2959.955	42	#002A	-0.548%	
G	3135.963	40	#0028	+0.350%	
G#	3322.438	38	#0026	+0.992%	
A	3520.000	36	#2924	+1.357%	
A#	3729.310	34	#0022	+1.417%	
B	3951.066	32	#0020	+1.134%	

Note	Frequency	Period		Error	Octave 4
C	4186.009	30	#001E	+0.462%	
C#	4434.922	28	#001C	-0.662%	
D	4698.636	27	#001B	+1.469%	
D#	4978.032	25	#0019	+1.441%	
E	5274.041	24	#0018	+1.246%	
F	5587.652	22	#0016	-1.685%	
F#	5919.911	21	#0015	-0.548%	
G	6271.927	20	#0014	+0.350%	
G#	6644.875	19	#0013	+0.992%	
A	7040.000	18	#0012	+1.357%	
A#	7458.621	17	#0011	+1.417%	
B	7902.133	16	#0010	+1.134%	

The notes in the scale of C major are given in a slightly more digestible form below.

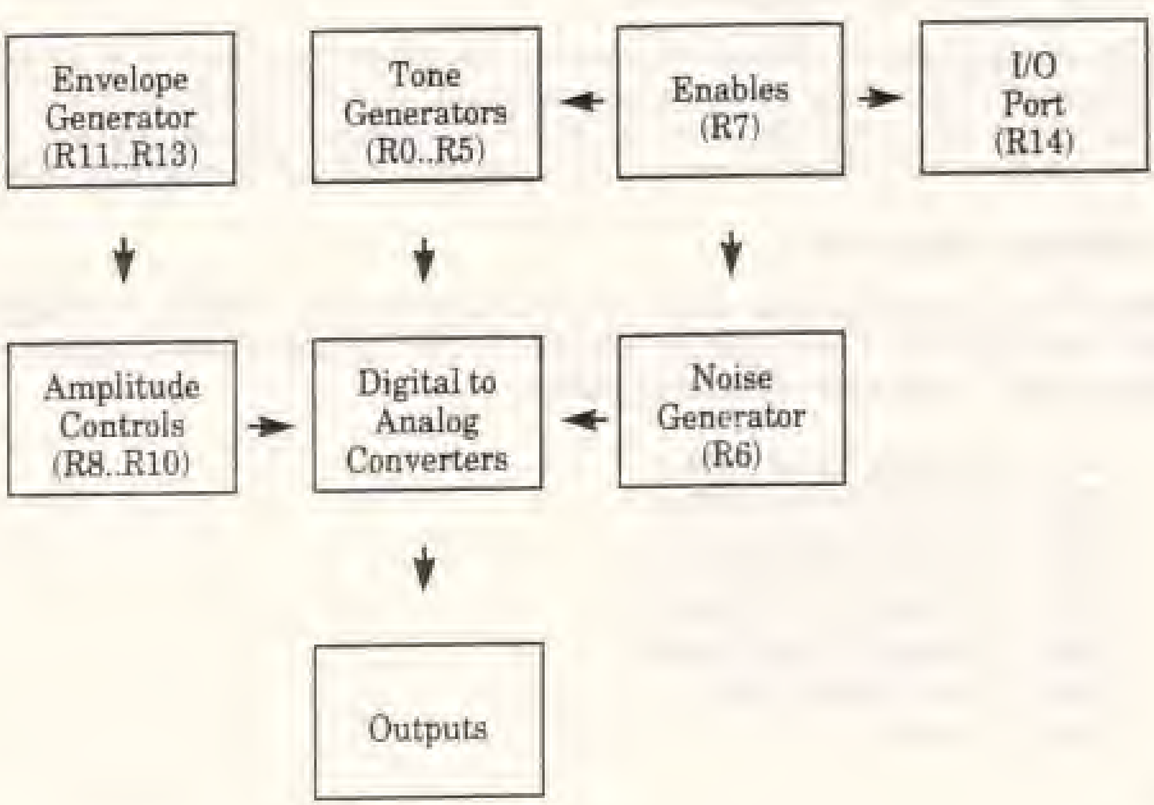


Appendix IX
The Programmable Sound Generator.

The programmable sound generator (PSG) is an AY-3-8912 chip. This is briefly described in section 7.1. The PSG has a number of registers which are described below. This information is provided for the interest of the user, particularly if hardware enveloping is to be used (in which case section (e) will be of special interest). However, the software enveloping provided by the Sound Manager can achieve all that the sound chip is capable of unless very short attacks or decays are required.

If the user is intending to drive the sound chip directly rather than by using the Sound Manager then the information presented is not complete and the user should consult the manufacturer's data sheet. The user is advised to call the routine MC SOUND REGISTER to write data to a sound chip register as this obeys the timing constraints on access to the sound chip.

The following diagram indicates the interactions between the various sections of the sound chip:



The sound chip data registers are as follows:

Register 0:	Channel A tone period fine tune.
Register 1:	Channel A tone period coarse tune.
Register 2:	Channel B tone period fine tune.
Register 3:	Channel B tone period coarse tune.
Register 4:	Channel C tone period fine tune.
Register 5:	Channel C tone period coarse tune.
Register 6:	Noise period.
Register 7:	Enables and I/O direction.
Register 8:	Channel A amplitude and envelope enable.
Register 9:	Channel B amplitude and envelope enable.
Register 10:	Channel C amplitude and envelope enable.
Register 11:	Envelope period fine tune.
Register 12:	Envelope period coarse tune.
Register 13:	Envelope shape.
Register 14:	Input from or output to port A.
Register 15:	Not used.

a. Tone Generators (Registers 0..5)

Each channel has two tone period registers associated with it. These set the period of the sound to be generated (in units of 8 microseconds) by that channel. The fine tune register stores the least significant 8 bits of the period; the coarse tune register stores the most significant 4 bits of the period. To include the tone in the output of a channel the appropriate bit in the enables register must be cleared.

b. Noise Generator (Register 6).

There is a single pseudo-random noise source. The output from this can be included in the output of any of the three channels (as specified by the enables register). The period of the noise generator is set by bits 0..4 of the noise period register. The period specifies the middle frequency of the noise produced in 8 microsecond units.

c. Enables (Register 7).

The enables register specifies whether tone or noise is to be included in the output from each channel. It also specifies whether the I/O port is to act in input or in output mode. The bits are allocated as follows:

Bit 0:	Channel A tone disable.
Bit 1:	Channel B tone disable.
Bit 2:	Channel C tone disable.
Bit 3:	Channel A noise disable.
Bit 4:	Channel B noise disable.
Bit 5:	Channel C noise disable.
Bit 6:	Port A output mode.
Bit 7:	Not used.

Note that port A is connected to the keyboard and joystick and so the port must always be in input mode. The user must ensure that bit 6 of the enables register is always set to zero.

d. Amplitude Controls (Registers 8..10).

Each channel has an amplitude control register associated with it. Bit 4 of this register specifies whether hardware enveloping is to be used for the channel. If the bit is set then the channel amplitude (volume) is under the control of the hardware envelope generator. If the bit is clear then the amplitude is set by bits 0..3 of the register - a value of 0 means no sound and a value of 15 means maximum volume.

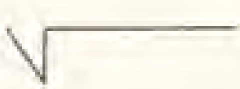
e. Envelope generator (Registers 11..13).

The sound chip has a single hardware envelope generator which can be used to control any combination of the three sound channels as specified by the channel's amplitude register (see (d) above). Bits 0 to 3 of register 13 control the shape of the envelope in a rather unobvious manner. The following table gives values required to generate each of the 8 hardware envelopes that are possible. Other values (0..7) duplicate envelopes 9 and 15.

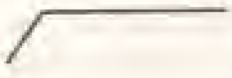
8:  Repeated jump up and ramp down.

9:  Jump up and ramp down once then hold at minimum volume (zero).

10:  Jump up then repeatedly ramp down and up again

11:  Jump up and ramp down then jump up and hold at maximum volume (fifteen).

12:  Repeatedly ramp up and drop down.

13:  Ramp up then hold at maximum volume (fifteen)

14:  Repeatedly ramp up and down again.

15:  Ramp up and drop down once then hold at minimum volume (zero).

The length of each of the ramps, upwards or downwards, is set by the envelope period. The envelope period is a full 16 bit value whose less significant byte is stored in register 11 and whose more significant byte is stored in register 12. The period is given in 128 microsecond units and is the time between steps in the ramp. Since the ramp has 16 steps (corresponding to the 16 volume settings) the total time taken for the ramp is the envelope period times 1024 microseconds (i.e. the envelope period approximately sets the total time for the ramp in milliseconds).

f. I/O Port (Register 14).

The mode of operation of the PSG port is set by a bit in the enables register (see section (c) above). However, since port A is dedicated to reading the keyboard and joysticks it should always be operated in input mode. The port may be read by reading the contents of register 14. However, scanning the keyboard is a complex action and is best left to the Key Manager which provides ample facilities for access to the keys.

References to port B in the manufacturer's data sheet should be ignored as the AY-3-8912 is a version of the chip that does not have port B.

Appendix X
Kernel Block Layouts.

The user provides a number of blocks to the Kernel for various puposes. The layouts of these blocks are described below, mainly for the interest of the user. There are very few occasions when the user is allowed to write to one of these blocks. Routines are provided to perform most actions that the user could wish to perform (see KL INIT EVENT, KL ADD TICKER, KL NEW FRAME FLY, KL NEW FAST TICKER and KL DISARM EVENT). These routines set values into the block from registers. The user should not write to the blocks, except as noted below.

All the following blocks must lie in the central 32K of RAM (otherwise the Kernel will be unable to access them).

a. Event Blocks.

See section 11 for a general discussion of events and event blocks. An event block is laid out as follows:

0,1:	Chain
2:	Count
3:	Class
4,5:	Routine address
6:	ROM
7+:	User fields

Chain is a system pointer which must never be written to by the user. It is used to store events on the various event queues.

Class records the type of the event. It should not be written to by the user.

Bit 0:	1 ⇒ Near address, 0 ⇒ Far address.
Bits 1..4:	Synchronous event priority.
Bit 5:	Must be zero.
Bit 6:	1 ⇒ Express event, 0 ⇒ Normal event.
Bit 7:	1 ⇒ Asynchronous event, 0 ⇒ Synchronous event.

Note that many system queues are kept in priority order and so the block must be requeued if the priority is changed, it is not sufficient merely to change the priority in the event block.

Count is the event count - a record of how many kicks are waiting to be processed or whether the event is disabled. See section 11.2 for a full discussion of the use of the event count.

Routine address and ROM make up the far address of the event routine. If the near address bit in the event class is true then the event routine is at a near address - the ROM select byte (byte 6) is ignored and the event routine is called directly. If the near address bit is false then the event routine is at far address - bytes 4,5 and 6 make up the far address to call to run the event routine. The user may write to the routine address and ROM fields (and to the near address bit in the class byte as well) provided that the operation is performed indivisibly (i.e. interrupts should be disabled).

The user fields are optional. They may be used to provide a data area specific to the event block so that a single event routine may be shared between a number of different event blocks (the event routine is passed the address of the user fields).

b. Ticker Queue Blocks.

See section 10 for a general discussion of ticker interrupts and the ticker queue. A ticker queue block is laid out as follows:

0,1:	Tick chain
2,3:	Tick count
4,5:	Recharge count
6+:	Event block

Tick chain is a system pointer which must never be written to by the user. It is used to store the block on the ticker queue.

Tick count is a count of the number of ticks before the next kick occurs. A tick count of zero means that the tick block is dormant and will not generate any kicks. (Ideally a dormant block should be removed from the ticker queue to avoid wasting time). The user may write to this field if required providing this is done indivisibly.

Recharge count is the value that the tick count is set to after each kick. If the recharge count is zero then the ticker block will become dormant after generating one kick. The user may write to this field if required providing this is done indivisibly.

Event block is a standard event block as described in section (a) above.

c. Frame Flyback Queue Blocks.

See section 10 for a general discussion of frame flyback interrupts and the frame flyback queue. A frame flyback queue block is laid out as follows:

0,1:	Frame chain
2+:	Event block

Frame chain is a system pointer which must never be written to by the user. It is used to store the block on the frame flyback queue.

Event block is a standard event block as described in section (a) above.

d. Fast Ticker Queue Blocks.

See section 10 for a general discussion of fast ticker interrupts and the fast ticker queue. A fast ticker queue block is laid out as follows:

0,1:	Fast chain
2+:	Event block

Fast chain is a system pointer which must never be written to by the user. It is used to store the block on the fast ticker queue.

Event block is a standard event block as described in section (a) above.

Appendix XI
The Alternate Register Set.

The Z80 microprocessor has two sets of registers - the normal set (AF, BC, DE and HL) and the alternate set (AF', BC', DE' and HL'). Unless the techniques outlined in this appendix are implemented the user is prohibited from using the alternate register set. This is because the alternate register set is used by the firmware (the Kernel in particular) for storing certain system values and flags. Providing that the user never enters the firmware then the alternate register set may be used without restriction. Of course this would mean that the user would be unable to use any facilities provided by the firmware. Furthermore, the user would also have to disable interrupts as interrupts cause firmware routines to be executed.

In the sections below a number of different methods are described that allow the user to overcome these restrictions. The method chosen will depend on what use is to be made of the alternate register set.

a. The firmware's use of the alternate register set.

The Kernel stores a couple of system variables in the alternate register set. This allows the Kernel to access these variables easily and thus speeds up a number of operations (particularly entry to and exit from firmware routines). Only BC' and the alternate carry flag (carry') are used to store values, however, routines do make use of the other alternate registers and so firmware routines may corrupt them.

B' is used to store the I/O address of the gate array (#7F). C' is used to store the value required to set the current ROM state and screen mode:

Bits 0..1:	Set the screen mode.
Bit 2:	Disables the lower ROM.
Bit 3:	Disables the upper ROM.
Bits 4..7:	System value to select gate array function.

By changing the ROM state bits and performing an OUT (C),C instruction the user can enable or disable ROMs. (N.B. The Z80 OUT (C),r and IN (C),r instructions use B as the top 8 bits of the I/O address. The hardware uses these top bits for decoding the I/O address, it ignores the bottom 8 bits!) OUT (C),C may be used to change the ROM state during the interrupt path when the normal Kernel entries (e.g. KL U ROM ENABLE) may not be called because they enable interrupts.

Carry' is normally false. When carry' is true this indicates that the firmware is in the interrupt path. The firmware occasionally uses this flag to allow it to take a different action when it is in the interrupt path to the action it takes when it is not in the interrupt path (usually avoiding enabling interrupts).

b. Simple use of the alternate register set.

The technique described in this section allows use of the alternate register set providing that no firmware routines are called and that interrupts are disabled.

After disabling interrupts registers A', DE' and HL' may be used as required. If registers BC' or F' (in particular carry') are used then their original contents must be restored before interrupts are re-enabled. The user may alter bits in C' (as described in (a) above) and need not restore the original value provided that an OUT (C),C is performed to keep the hardware abreast of the current state. The machine will not function correctly if the hardware and the value in C' are out of step when interrupts are enabled.

This technique requires interrupts to be disabled for the duration of the operation being performed. This is acceptable if the operation is short but not if the operation is lengthy. Disabling interrupts for a lengthy period will stop many firmware functions such as timers (and hence ink flashing, sound generation and keyboard scanning). If the operation to be performed is lengthy then it might be better to consider the use of one of the techniques described in sections (c) or (d) instead.

Example.

The user might want to provide a routine that performs an LD A,(BC) from RAM (similar to the RAM LAM pseudo-instruction provided by the firmware).

The code for this routine could be written as follows:

```
A_FROM_BC:
        PUSH    BC
        DI                      ;** About to use alternate registers
        EXX
        POP     HL              ;Transfer BC to HL'
;
        LD      A, C
        SET     2, A            ;Set the disable lower ROM bit
        SET     3, A            ;Set the disable upper ROM bit
        OUT     (C), A          ;Tell the hardware
;
        LD      A, (HL)         ;Read the value from RAM
;
        OUT     (C), C          ;Restore the old ROM state
;
        EXX
        EI                      ;** End of use of alternate registers
        RET
```

N.B. This routine must be RAM resident or disabling the ROMs will have an unfortunate effect!

c. Use of the alternate register set with interrupts enabled.

The technique described in this section allows the alternate register set to be used and interrupts to be enabled. It does not allow firmware routines to be called.

The simplistic use of the alternate register set by disabling interrupts as described above is unsatisfactory if this results in interrupts being disabled for an extended period of time. By patching INTERRUPT ENTRY in the low Kernel jumpblock interrupts can be trapped and appropriate action to restore the firmware registers can be taken. The actions that must be performed are as follows:

> Before starting to use the alternate register set the firmware's BC' is saved and INTERRUPT ENTRY is patched so that the user's interrupt routine is used.
>
> When the user has finished with the alternate register set the firmware's BC' and carry' are restored and INTERRUPT ENTRY is patched back to the firmware's interrupt routine.
>
> When an interrupt occurs the user's alternate registers are saved, the firmware's BC' and carry' are restored and INTERRUPT ENTRY is patched back to the firmware's interrupt routine. The latter is done in case a second interrupt occurs whilst processing the events kicked from the interrupt path of the first interrupt (remember that the event processing is performed with interupts enabled).
>
> After interrupt processing has finished the firmware's BC' is saved, the user's alternate registers are restored and INTERRUPT ENTRY is patched back to the user's interrupt routine again.

Note that when INTERRUPT ENTRY is patched it is vital to ensure that the lower ROM is disabled and remains disabled. It is impossible to patch the ROM version of INTERRUPT ENTRY! If an interrupt occured whilst the lower ROM was enabled then the firmware would jump straight into its interrupt routine without restoring its alternate registers first.

Example.

The following routines implement the scheme described above:

```
;
; The following storage locations are used
;
FIRM_BC:   DEFS 2        ;Two bytes to store the firmware's BC'
FIRM_INT:  DEFS 2        ;Two bytes to store the address of the
                         ;firmware's interrupt routine
USER_AF:   DEFS 2        ;Two bytes to store the user's AF'
USER_BC:   DEFS 2        ;Two bytes to store the user's BC'
USER_DE:   DEFS 2        ;Two bytes to store the user's DE'
USER_HL:   DEFS 2        ;Two bytes to store the user's HL'
```

```
;
;    This routine sets up the environment so that the
;    user may make use of the alternate register set.
;    N.B. Interrupts are enabled.
;
;
;
USER_ALTERNATE:
    DI                                  ;An interrupt would be disastrous
    EX      AF, AF'
    EXX                                 ;Swap to the alternate register set
;
    LD      (FIRM_BC), BC               ;Save the firmware's BC'
;
    LD      HL,(INTERUPT_ENTRY+1)
    LD      (FIRM_INT), HL              ;Save the firmware's interrupt routine
;
    LD      HL, USER_INTERRUPT          ;Use the replacement interrupt routine
    LD      (INTERRUPT_ENTRY + 1), HL
;
    LD      HL, (USER_AF)
    PUSH    HL
    POP     AF                          ;Restore user's AF'
    LD      BC, (USER_BC)               ;Restore user's BC'
    LD      DE, (USER_DE)               ;Restore user's DE'
    LD      HL, (USER_HL)               ;Restore user's HL'
;
    EXX                                 ;Swap back to the standard register set
    EX      AF, AF'
    EI                                  ;We have finished with the alternate regs
    RET
;
;    This routine restores the environment for the
;    firmware to use the alternate register set.
;    N.B. Interrupts are disabled and not re-enabled.
;
;
;
FIRM_ALTERNATE:
    DI                                  ;An interrupt would be disastrous
    EX      AF, AF'
    EXX                                 ;Swap to the alternate register set
;
    LD      (USER_HL), HL               ;Save user's HL'
    LD      (USER_DE), DE               ;Save user's DE'
    LD      (USER_BC), BC               ;Save user's BC'
    PUSH    AF
    POP     HL
    LD      (USER_AF), HL               ;Save user's AF'
;
```

```
    LD      HL, (FIRM_INT)           ;Restore the firmware's interrupt routine
    LD      (INTERRUPT_ENTRY + 1), HL

    LD      BC, (FIRM_BC)            ;Restore the firmware's BC'
    OR      A, A                     ;Set the firmware's carry' to be false

    EXX                              ;Swap back to the standard register set
    EX      AF, AF'
    RET                              ;N.B. May be about to enter the interrupt
;                                    ;path so no EI.
;
;
;   This routine replaces the firmware's interrupt routine
;   when the user is using the alternate register set
;
;
;
USER_INTERRUPT:
    CALL    FIRM_ALTERNATE           ;Switch the environment to the firmware
    CALL    INTERRUPT_ENTRY          ;Run the normal interrupt routine
    JP      USER_ALTERNATE           ;Switch the environment back to the user
```

To start using the alternate register set the user obeys the instruction:

```
CALL  USER_ALTERNATE
```

To finish using the alternate register set the user obeys the instructions:

```
CALL  FIRM_ALTERNATE
EI
```

d. Calling firmware routines whilst using the alternate register set.

The technique described in this section extends the technique described in section (c) to allow the user to call firmware routines whilst using the alternate register set.

To call a firmware routine requires exactly the same action as is required for the interrupt routine:

> Before calling a firmware routine the user's alternate registers are saved, the firmware's BC' and carry' are restored and INTERRUPT ENTRY is patched back to the firmware's interrupt routine. The latter is done in case an interrupt occurs whilst executing the firmware routine.
>
> After running the firmware routine the firmware's BC' is saved, the user's alternate registers are restored and INTERRUPT ENTRY is patched back to the user's interrupt routine again.

As indicated in section (c) it is vital to ensure that the lower ROM remains disabled while the alternate register set is in use since INTERRUPT ENTRY in the ROM is not patchable and jumps straight to the firmware's interrupt routine.

Using the routines defined in section (c) a firmware routine may be called by using the following sequence:

```
  ..
  CALL   FIRM_ALTERNATE          ;Switch the environment to the firmware
  EI                             ;FIRM ALTERNATE disables interrupts
  CALL   firmware                ;Run the firmware routine
  CALL   USER_ALTERNATE          ;Switch the environment back to the user
  ..
```

The above code is rather long if a lot of firmware calls are to be made (10 bytes per call). The following routine takes the address of a firmware routine to call to as an inline parameter (and only uses 5 bytes per call).

```
;
;     This routine saves the user's alternate registers, calls a
;     firmware routine whose address is passed inline and then
;     restores the user's alternate register set afterwards.
;
;
FIRM_ROUTINE:
        CALL   FIRM ALTERNATE          ;Switch to the firmware environment
;
        EXX                            ;N.B. Interrupts are disabled
        POP    HL                      ;Recover address of routine to call, uses
        LD E,  (HL)                    ;firmware's DE' and HL' which may be
        INC    HL                      ;corrupted
        LD D,  (HL)                    ;Get routine to call into DE'
        INC    HL
        PUSH HL                        ;Put the real return address back
;
        LD     HL, USER ALTERNATE      ;Restore the user environment when
                                       ;the firmware returns by putting a
        PUSH HL                        ;fake return address on the stack
;
        PUSH DE                        ;Save the routine to call
        EXX
        EI
        RET                            ;Jump to the routine to call
```

To call a firmware routine using the above routine the following sequence should be used:

```
  ..
  CALL   FIRM_ROUTINE
  firmware                       ;Address of routine to call
  ..                             ;FIRM ROUTINE returns here
```

Appendix XII
The Hardware.

A. Processor.

The processor is a Z80A running at a clock frequency of 4.00 MHz (±0.1%). There is logic that stretches $\overline{\text{MREQ}}$ and $\overline{\text{IORQ}}$ using the $\overline{\text{CPU WAIT}}$ facility so that the processor can only make one access to memory each microsecond.

The processor $\overline{\text{NMI}}$ pin is pulled up and made available on the expansion bus. However, a non-maskable interrupt may cause the firmware to violate various timing constraints and so its use is not recommended.

The processor interrupt pin is driven by a flip-flop in the video gate array. This flip-flop is set during every vertical flyback and every 52 scan lines thereafter until the next vertical flyback. The interrupt is arranged to occur approximately 2 scans (125 microseconds) into the 8 scan (500 microsecond) vertical flyback signal. The interrupt latch is cleared by the processor acknowledging the interrupt, or explicitly, using a software command. The top bit of the divide by 52 scan counter is also cleared when the processor acknowledges an interrupt occurring after this counter has overflowed. This allows the interrupt system to be expanded.

B. Memory.

ROM

A single 32K byte ROM is present on the processor board, but is mapped onto two blocks of 16K in processor address space. The lower half of the ROM occupies addresses #0000 to #3FFF and the upper half occupies addresses #C000 to #FFFF. These two halves of the ROM can be separately enabled and disabled by two control latches in the video gate array. On power-up or other system reset both halves of the ROM are enabled.

An expansion port signal can be used to disable this internal ROM and allow external ROM(s) to be accessed instead. These are selected by output instructions and replace the upper half of the on-board 32K byte ROM when selected.

RAM

64K bytes of dynamic RAM are fitted to the processor board at addresses #0000 to #FFFF. The lowest 16K and the top 16K are overlayed when ROM is enabled. Whether the ROM is enabled or not affects where data is read from, it has no effect on write operations which will be correctly performed 'through' the enabled ROM to the underlying RAM.

VDU SCREEN MEMORY

The display uses 16k of the processor RAM memory as screen refresh memory. The 16k used can be switched between the blocks starting at #0000, #4000, #8000 and at #C000 by the top two bits (bits 12 and 13) programmed into the HD6845S start address register (see section 6.4 for further details).

The arrangement of data in the VDU screen memory is dependent on the VDU mode currently selected. In all modes the memory can be considered as consisting of 8K 16 bit words. Each word contains either 4, 8 or 16 pixels (P0..Pn) of 1, 2 or 4 bits (B0..Bm) depending on the mode as follows:

A0	Bit	Mode 0	Mode 1	Mode 2
0	D7	P0 B0	P0 B0	P0 B0
0	D6	P1 B0	P1 B0	P1 B0
0	D5	P0 B2	P2 B0	P2 B0
0	D4	P1 B2	P3 B0	P3 B0
0	D3	P0 B1	P0 B1	P4 B0
0	D2	P1 B1	P1 B1	P5 B0
0	D1	P0 B3	P2 B1	P6 B0
0	D0	P1 B3	P3 B1	P7 B0
1	D7	P2 B0	P4 B0	P8 B0
1	D6	P3 B0	P5 B0	P9 B0
1	D5	P2 B2	P6 B0	P10 B0
1	D4	P3 B2	P7 B0	P11 B0
1	D3	P2 B1	P4 B1	P12 B0
1	D2	P3 B1	P5 B1	P13 B0
1	D1	P2 B3	P6 B1	P14 B0
1	D0	P3 B3	P7 B1	P15 B0

Data for lines 0,8,16,24.. on the display are packed into the first 2K byte block of the memory, lines 1,9,17,25.. are packed into the corresponding places of the next 2K byte block of memory, with lines 7,15,23,31.. occupying the top 2K byte block of the 16k memory area.

The bottom 10 bits of the HD6845SP start address register define where within these 2K blocks the screen starts. The offset from the start of the 2K byte block is always even and is calculated as twice the register contents modulo 2K bytes. When data has to be displayed from beyond the end of a 2K byte block wrap around occurs to the begining of the same 2K byte block. See section 6.4 for a fuller description.

D. AY-3-8912 Programmable Sound Generator.

The PSG is accessed using ports A and C of the µPD8255 device. Note that when writing or loading address to the AY-3-8912 the maximum duration of the write or load address command with BDIR high is 10 microseconds. The clock input to the sound generator is exactly 1.00 MHz. The BC2 signal is tied permanently high. On power-up the I/O port should be programmed to input mode.

The user is advised to use the firmware routine MC SOUND REGISTER to write to the PSG.

E. HD6845S CRT Controller (HD6845S CRTC).

The character clock to the CRTC occurs for every two bytes fetched from memory, i.e. every 1.0 microseconds. The first byte of a pair has an even address, the second has an odd address. In normal operation the internal registers should be set up as follows:

Register	Function	PAL	SECAM	NTSC
0	Horizontal Total	63	63	63
1	Horizontal Displayed	40	40	40
2	Horizontal Sync. Posn.	46	46	46
3	Vsync., Hsync. widths	#8E	#8E	#8E
4	Vertical Total	38	38	31
5	Vertical Total Adjust	0	0	6
6	Vertical Displayed	25	25	25
7	Vertical Sync. Posn.	30	30	27
8	Interlace and Skew	0	0	0
9	Max. Raster Address	7	7	7
10	Cursor Start Raster	X	X	X
11	Cursor End Raster	X	X	X
12	Start Address (H)	X	X	X
13	Start Address (L)	X	X	X
14	Cursor (H)	X	X	X
15	Cursor (L)	X	X	X

In the above table the numbers for PAL and SECAM standards are identical.

Note that X indicates that software may vary these numbers during device operation. The firmware only makes use of the start address register which is used to set the screen base and offset.

F. Video Gate Array.

The software must access this device in order to control the enabling and disabling of ROMs, the mode of operation of the VDU and also to load colour information for 'inks' into the palette memory. One I/O channel is used for all commands, the top two bits of data specifying the command type as follows:

Bit 7	Bit 6	Use
0	0	Load palette pointer register.
0	1	Load palette memory.
1	0	Load mode and ROM enable register.
1	1	Reserved.

MODE AND ROM ENABLE REGISTER

This write-only register controls the VDU mode and ROM enabling as follows:

Bit 7:	1
Bit 6:	0
Bit 5:	** Reserved ** (send 0)
Bit 4:	Clear raster 52 divider.
Bit 3:	Upper half ROM disable.
Bit 2:	Lower half ROM disable.
Bit 1:	VDU Mode control MC1.
Bit 0:	VDU Mode control MC0.

Writing a 1 to bit 4 clears the top bit of the divide by 52 counter used for generating periodic interrupts.

Modes are defined by the mode control pins as follows:

MC1	MC0	Mode
0	0	Mode 0, 160 x 200 pixels in 16 colours
0	1	Mode 1, 320 x 200 pixels in 4 colours.
1	0	Mode 2, 640 x 200 pixels in 2 colours.
1	1	** Do not use **

The gate array hardware synchronises mode changing to the next horizontal fly-back in order to aid software that requires different parts of the screen to be handled in different modes.

On power-up and other system resets, the mode and ROM enable register is set to zero, enabling both halves of the ROM.

PALETTE POINTER REGISTER

This write-only register controls the loading of the VDU colour palette as follows:

Bit 7:	0
Bit 6:	0
Bit 5:	** Reserved ** (send 0)
Bit 4:	Palette pointer bit PR4.
Bit 3:	Palette pointer bit PR3.
Bit 2:	Palette pointer bit PR2.
Bit 1:	Palette pointer bit PR1.
Bit 0:	Palette pointer bit PR0.

Bits PR0 to PR3 select which ink is to have its colour loaded, providing bit PR4 is low. If bit PR4 is high then bits PR0 to PR3 are ignored and the border ink colour is loaded.

PALETTE MEMORY

This write-only memory controls the VDU colour palette as follows:

Bit 7:	0
Bit 6:	1
Bit 5:	** Reserved ** (send 0)
Bit 4:	Colour data bit CD4.
Bit 3:	Colour data bit CD3.
Bit 2:	Colour data bit CD2.
Bit 1:	Colour data bit CD1.
Bit 0:	Colour data bit CD0.

The ink entry pointed at by the palette pointer register is loaded with the colour sent on this channel. The number of colours that need to be loaded ranges from 2 colours in mode 2 to 16 colours in mode 0. In addition to loading the colours an extra colour data byte must be sent to this channel to define the border colour. On power-up and other system resets the contents of the palette are undefined, but the border colour is set to BLACK, to avoid unsightly effects on power-up.

The 32 colour codes are decoded to drive the RGB signals, producing 27 different colours. The hardware colours are listed in Appendix V.

G. μPD8255 Parallel Peripheral Interface.

The PPI as well as the 8 port pins on the PSG are used to interface to the keyboard and to control and sense miscellaneous signals on the processor board. Port A must be programmed either to input or to output in mode 0 since this port is used for reading and writing to the PSG. Port B must be programmed to input in mode 0. Port C must be programmed to output in mode 0 on both halves.

Circuitry is provided around the PPI to reset it during system reset. For details of the operation of the μPD8255 see the NEC product specification.

CHANNEL A (Input or Output)

Bit 7:	Data/Address DA7 connected to AY-3-8912.
Bit 6:	Data/Address DA6 connected to AY-3-8912.
Bit 5:	Data/Address DA5 connected to AY-3-8912.
Bit 4:	Data/Address DA4 connected to AY-3-8912.
Bit 3:	Data/Address DA3 connected to AY-3-8912.
Bit 2:	Data/Address DA2 connected to AY-3-8912.
Bit 1:	Data/Address DA1 connected to AY-3-8912.
Bit 0:	Data/Address DA0 connected to AY-3-8912.

CHANNEL B (Input Only)

Bit 7:	Datacorder cassette read data.
Bit 6:	Centronics busy signal.
Bit 5:	Not expansion port active signal.
Bit 4:	Not option link LK4.
Bit 3:	Not option link LK3.
Bit 2:	Not option link LK2.
Bit 1:	Not option link LK1.
Bit 0:	Frame flyback pulse.

The option links, LK1..LK4 are factory set. LK4 is fitted for 60 Hz T.V. standards and omitted for 50 Hz standards.

CHANNEL C (Output Only)

Bit 7:	AY-3-8912 BDIR signal.
Bit 6:	AY-3-8912 BC1 signal.
Bit 5:	Datacorder cassette write data
Bit 4:	Datacorder cassette motor on.
Bit 3:	Keyboard row select KR3.
Bit 2:	Keyboard row select KR2.
Bit 1:	Keyboard row select KR1.
Bit 0:	Keyboard row select KR0.

H. Centronics Port Latch.

This latch is loaded with data by output commands to the correct I/O channel. It cannot be read. Note that the timing requirements on Centronics interfaces generally specify that the data must be present on the seven data lines at least 1 microsecond before the strobe is made active and must remain valid for at least 1 microsecond after the strobe returns inactive. The duration of the strobe must be between 1 and 500 microseconds. The busy signal can be inspected as soon as the strobe is inactive in order to determine when more data can be sent.

Bit 7:	Centronics strobe signal (1 = active).
Bit 6:	Data 7 to Centronics port.
Bit 5:	Data 6 to Centronics port.
Bit 4:	Data 5 to Centronics port.
Bit 3:	Data 4 to Centronics port.
Bit 2:	Data 3 to Centronics port.
Bit 1:	Data 2 to Centronics port.
Bit 0:	Data 1 to Centronics port.

On power-up and other system resets the outputs of this latch are all cleared.

I. Keyboard and Joysticks.

The keyboard and joystick switches are sensed by selecting one of ten rows using the four control bits on channel C of the PPI and reading the data from the PSG parallel port using port A of the PPI.

The keyboard and joystick switches are arranged in a 10 by 8 matrix. One of ten rows is selected using the code on KR0..KR3 and the eight bits of data are then read in parallel on a parallel port as described above. A switch is active (closed) if the corresponding data bit is a logic 0.

The key number associated with each key (see Appendix I) is constructed as follows:

Bit:	7	6	5	4	3	2	1	0
	0	Row number				Bit number		

Thus the key that is associated with bit 5 in row 4 has key number 37 (4*8+5).